Pro Git 第二版（中文版）

看云文档小组



目　录

前言
- Scott Chacon 序
- Ben Straub 序
- 献辞
- 贡献者
- 引言

1. 起步
 - 1.1 关于版本控制
 - 1.2 Git 简史
 - 1.3 Git 基础
 - 1.4 命令行
 - 1.5 安装 Git
 - 1.6 初次运行 Git 前的配置
 - 1.7 获取帮助
 - 1.8 总结
2. Git 基础
 - 2.1 获取 Git 仓库
 - 2.2 记录每次更新到仓库
 - 2.3 查看提交历史
 - 2.4 撤消操作
 - 2.5 远程仓库的使用
 - 2.6 打标签
 - 2.7 Git 别名
 - 2.8 总结
3. Git 分支
 - 3.1 分支简介
 - 3.2 分支的新建与合并
 - 3.3 分支管理
 - 3.4 分支开发工作流
 - 3.5 远程分支
 - 3.6 变基
 - 3.7 总结
4. 服务器上的 Git
 - 4.1 协议
 - 4.2 在服务器上搭建 Git
 - 4.3 生成 SSH 公钥
 - 4.4 配置服务器
 - 4.5 Git 守护进程
 - 4.6 Smart HTTP

4.7 GitWeb
4.8 GitLab
4.9 第三方托管的选择
4.10 总结
5. 分布式 Git
5.1 分布式工作流程
5.2 向一个项目贡献
5.3 维护项目
5.4 总结
6. GitHub
6.1 账户的创建和配置
6.2 对项目做出贡献
6.3 维护项目
6.4 管理组织
6.5 脚本 GitHub
6.6 总结
7. Git 工具
7.1 选择修订版本
7.2 交互式暂存
7.3 储藏与清理
7.4 签署工作
7.5 搜索
7.6 重写历史
7.7 重置揭密
7.8 高级合并
7.9 Rerere
7.10 使用 Git 调试
7.11 子模块
7.12 打包
7.13 替换
7.14 凭证存储
7.15 总结
8. 自定义 Git
8.1 配置 Git
8.2 Git 属性
8.3 Git 钩子
8.4 使用强制策略的一个例子
8.5 总结
9. Git 与其他系统
9.1 作为客户端的 Git

9.2 迁移到 Git
9.3 总结
10. Git 内部原理
10.1 底层命令和高层命令
10.2 Git 对象
10.3 Git 引用
10.4 包文件
10.5 引用规格
10.6 传输协议
10.7 维护与数据恢复
10.8 环境变量
10.9 总结
A. 其它环境中的 Git
A1.1 图形界面
A1.2 Visual Studio 中的 Git
A1.3 Eclipse 中的 Git
A1.4 Bash 中的 Git
A1.5 Zsh 中的 Git
A1.6 Powershell 中的 Git
A1.7 总结
B. 将 Git 嵌入你的应用
A2.1 命令行 Git 方式
A2.2 Libgit2
A2.3 JGit
C. Git 命令
A3.1 设置与配置
A3.2 获取与创建项目
A3.3 快照基础
A3.4 分支与合并
A3.5 项目分享与更新
A3.6 检查与比较
A3.7 调试
A3.8 补丁
A3.9 邮件
A3.10 外部系统
A3.11 管理
A3.12 底层命令

前言

> 原文出处：http://www.git-scm.com/book/

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Scott Chacon 序

Scott Chacon 序

欢迎来到 Pro Git 第二版。第一版出版到现在已经过去了四年。到今天，Git 虽然出现了许多改变，但是还有很多重要的事情一如昨日。因为 Git 核心团队对保持向后兼容性异常固执，所以直到今天大多数核心命令与概念依然有效，但是围绕 Git 的社区还是有一些重大的增加与改变。本书的第二版就是为了更新书籍并讲解那些改动以使其对新用户更有帮助。

当我写第一版时，Git 对于超级黑客来说还是一个相对难用，只能勉强接受的工具。它开始在特定的社区中快速发展，但是还没有达到像今天一样无处不在的地步。自那时起，几乎每一个开源社区都采用了它。Git 在 Windows 上取得了难以置信的进步，包括所有平台的图形用户界面对它的支持、IDE 的支持，以及商业使用的爆炸式发展。四年前的 Pro Git 对此一无所知。新版本的主要目标之一就是涉及 Git 社区中那些所有新的前沿领域。

使用 Git 的开源社区也呈现出爆炸式的发展。大概在五年前吧，我坐下来写这本书时（写完第一个版本花了我不少时间），我开始在一个知名度极小的开发 Git 托管网站的公司工作，这家公司就是 GitHub。本书出版时大概有几千人在使用 GitHub 网站，而为其工作的只有我们四个人。在我写这篇介绍时，GitHub 宣布我们托管了 1000 万个项目、拥有大概 500 万注册开发者账户与大概 230 名员工。爱它也好，恨它也罢，当我坐下来写第一版时，GitHub 以一种意想不到的方式猛烈地改变了一大批开源社区。

我在 Pro Git 的原始版本中写了一节我并不是很满意的内容，是作为和提供 Git 托管服务相关的例子的 GitHub。我在书里写的东西本质上都是和社区有关的，但是又不得不讨论到我的公司，这点我不喜欢。同时我还不喜欢那个兴趣的冲突，GitHub 在 Git 社区中的重要性是无法避免的。我已经决定将本书的那部分转变为深度介绍 GitHub 是什么以及如何高效地使用它，而不再是作为一个 Git 托管的例子。如果你正学习如何使用 Git，那么了解如何使用 GitHub 将会帮助你加入到一个巨大的社区中。不论你决定为自己的代码使用哪一个 Git 托管服务，这都很有价值。

自从上次出版以来另一个重大变革是 Git 网络传输 HTTP 协议的开发与崛起。书中的大多数例子都已经从 SSH 切换到 HTTP，因为它更简单。

在过去这几年看到 Git 从一个相对无名的版本管理系统成长为商业与开源版本管理的事实标准是令人吃惊的。我很高兴 Pro Git 做得很好并已经成为市场上几本既成功又完全开源的技术书籍之一。

我希望你能享受这个升级版的 Pro Git。

Ben Straub 序

Ben Straub 序

本书的第一版就是将我与 Git 结下不解之缘的原因。书中采用的是我引进的做软件的风格，这种风格比我之前看到的任何事情都要自然。那时我已经做了好几年开发者了，但是这本书将我指引到一条更加精彩的道路上。

几年之后的现在，我是 Git 的一个主要实现的贡献者，我在最大的 Git 托管公司工作，我已经环游世界教人们使用 Git。当 Scott 问我是否有兴趣在第二版上工作时，我甚至连想都没想就答应了。

能在这本书上工作是一份巨大的快乐与荣耀。我希望它能像帮助我一样帮助你。

献辞

献辞

致我的妻子，Becky，没有她的话这段冒险不会开始。— Ben

谨以此书献给我的家人。给这些年一直支持着我的妻子 Jessica 和女儿 Josephine，还有那些在我风烛残年之时还能支持我的人。— Scott

贡献者

贡献者

因为这是一本开源书籍，这几年我们获得了捐赠的若干勘误表与内容修改。下面是将 ProGit 英文版作为开源项目贡献的所有人员。感谢你们帮助将这本书变成对每一个人都更好的书。

```
Aaron Schumacher
Aggelos Orfanakos
Alec Clews
Alex Moundalexis
Alexander Harkness
Alexander Kahn
Andrew McCarthy
AntonioK
Benjamin Bergman
Brennon Bortz
Brian P O'Rourke
Bryan Goines
Cameron Wright
Chris Down
Christian Kluge
Christoph Korn
Ciro Santilli
Cor
Dan Croak
Dan Johnson
Daniel Kay
Daniel Rosen
DanielWeber
Dave Dash
Davide Fiorentino lo Regio
Dilip M
Dimitar Bonev
Emmanuel Trillaud
Eric-Paul Lecluse
Eugene Serkin
Fernando Dobladez
Gordon McCreight
Helmut K. C. Tessarek
Igor Murzov
Ilya Kuznetsov
Jason St. John
Jay Taggart
Jean Jordaan
Jean-Noël Avila
Jean-Noël Rouvignac
Jed Hartman
Jeffrey Forman
John DeStefano
Junior
Kieran Spear
Larry Shatzer, Jr
Linquize
Markus
Matt Deacalion Stevens
Matthew McCullough
```

```
Matthieu Moy
Max F. Albrecht
Michael Schneider
Mike D. Smith
Mike Limansky
Olivier Trichet
Ondrej Novy
Ori Avtalion
Paul Baumgart
Peter Vojtek
Philipp Kempgen
Philippe Lhoste
PowerKiKi
Radek Simko
Rasmus Abrahamsen
Reinhard Holler
Ross Light
Ryuichi Okumura
Sebastian Wiesinger
Severyn Kozak
Shane
Shannen
Sitaram Chamarty
Soon Van
Sven Axelsson
Tim Court
Tuomas Suutari
Vlad Gorodetsky
W. Trevor King
Wyatt Carss
Włodzimierz Gajda
Xue Fuqiao
Yue Lin Ho
adelcambre
anaran
bdukes
burningTyger
cor
iosias
nicesw123
onovy
pcasaretto
sampablokuper
```

引言

引言

你将花费你生命中的若干小时来阅读有关 Git 的相关内容。让我们用几分钟时间来介绍下我们将给你讲解的内容。下面是本书正文十章和附录三章的快速总结。

在第一章，我们将介绍版本控制系统（VCSs）和 Git 的基本概念——不涉及技术内容，仅仅是什么是 Git，为什么它会成为 VCSs 大家庭中的一员，它与其它 VCSs 的区别，以及为什么那么多人都在使用 Git。然后，我们将介绍如何下载 Git 以及如果你的系统没有安装 Git，如何为第一次运行做准备。

在第二章，我们将阐述 Git 的基本使用——包含你在使用 Git 时可能遇到的 80% 的情形。通过阅读本章，你应该能够克隆仓库、查看项目历史、修改文件和贡献更改。如果本书在此刻自燃，你应该已经能够使用已经学到的漂亮有用的 Git 知识获取到另外一份拷贝。

第三章关注于 Git 的分支模型。分支模型通常被认为是 Git 的杀手级特性。这里，你将学习到究竟是什么让 Git与众不同。学习完本章，你可能需要一段时间来思考，在 Git 分支成为你的生活的一部分之前，你到底是如何生活的。

第三章关注于服务器端的 Git。本章面向那些希望在你自己的组织或个人服务器搭建用于合作的 Git 的读者。如果你希望让别人处理这些事务，我们也会探讨一些托管选项。

第五章将阐述多种分布式工作流的细节，以及如何使用 Git 实现它们。学习完本章，你应该能够在多个远程仓库之间游刃有余，通过电子邮件使用 Git，熟练地处理多个远程分支和合作者贡献的补丁。

第六章介绍 GitHub 托管服务以及深层次的工具。我们将涵盖注册与账户管理，创建和使用 Git 仓库，贡献项目的普通工作流以及接受他人的贡献，GitHub 的可编程接口和那些能够让你的生活变得更简单的小技巧。

第七章关于 Git 的高级命令。你将学习到一些高级主题，诸如掌握可怕的“reset”命令，使用二分搜索识别错误，编辑历史，细节版本选择等等。本章的介绍将丰富你的 Git 知识，让你成为一个真正的大师。

第八章关于 Git 环境的自定义配置，包括设置用于增强或促进自定义策略的钩子脚本以及按照你所需要的方式进行工作的环境配置。我们还会介绍构建你自己的脚本集，以增强自定义提交策略。

第九章对比 Git 和其它 VCSs，包括在 Subversion（SVN）的世界使用 Git 以及从其它 VCSs 迁移到 Git。很多组织仍在使用 SVN，并且也没有计划改变，此时，你将了解到 Git 不可思议的能力——本章将展示，在你不得不使用 SVN 服务器的时候如何协同合作。我们还将介绍如何从不同系统导入项目，以便你能够全身心投入 Git 的怀抱。

第十章深入 Git 阴暗而漂亮的实现细节。现在，你已经知道所有有关 Git 的知识，能够熟练运用 Git 的强大优雅的功能。接下来，你可以继续学习 Git 如何存储对象、Git 的对象模型是怎样的、打包文件的细

节、服务器协议等更多知识。本书自始至终都将引用本章的内容，以便你能够在当时就可以深入了解。但是，如果你像我们一样希望深入学习技术细节，你可能想先阅读第十章。我们将选择权交给你。

在附录 A，我们学习多个在特定环境中使用 Git 的实例。我们涵盖多个你可能会使用 Git 的多个 GUI 和 IDE 编程环境，这些都可以由你自己选择。如果你想在 shell、Visual Studio 或 Eclipse 中使用 Git，请阅读本章。

在附录 B，我们探讨通过类似 libgit2 和 JGit 的工具编写 Git 脚本、扩展 Git。如果你对编写复杂、快速的自定义工具感兴趣，需要了解 Git 的底层访问，本章就是你所需要了解的。

最后在附录 C，我们一次性浏览 Git 的所有主要命令，复习在本书中介绍的内容，回忆我们能够使用这些命令做什么。如果你需要知道本书中我们使用了哪些特定 Git 命令，你可以在这里查阅。

下面让我们开始。

1. 起步

第1章. 起步

本章关于开始学习 Git。我们从介绍有关版本控制工具的一些背景知识开始，然后讲解如何在你的系统运行 Git，最后是关于如何设置 Git 开始你的工作。通过本章的学习，你应该了解为什么 Git 这么流行，为什么你应该使用 Git 以及你应该如何设置以便使用 Git。

1.1 关于版本控制

关于版本控制

什么是“版本控制”？我为什么要关心它呢？ 版本控制是一种记录一个或若干文件内容变化，以便将来查阅特定版本修订情况的系统。 在本书所展示的例子中，我们对保存着软件源代码的文件作版本控制，但实际上，你可以对任何类型的文件进行版本控制。

如果你是位图形或网页设计师，可能会需要保存某一幅图片或页面布局文件的所有修订版本（这或许是你非常渴望拥有的功能），采用版本控制系统（VCS）是个明智的选择。 有了它你就可以将某个文件回溯到之前的状态，甚至将整个项目都回退到过去某个时间点的状态，你可以比较文件的变化细节，查出最后是谁修改了哪个地方，从而找出导致怪异问题出现的原因，又是谁在何时报告了某个功能缺陷等等。 使用版本控制系统通常还意味着，就算你乱来一气把整个项目中的文件改的改删的删，你也照样可以轻松恢复到原先的样子。 但额外增加的工作量却微乎其微。

本地版本控制系统

许多人习惯用复制整个项目目录的方式来保存不同的版本，或许还会改名加上备份时间以示区别。 这么做唯一的好处就是简单，但是特别容易犯错。 有时候会混淆所在的工作目录，一不小心会写错文件或者覆盖意想外的文件。

为了解决这个问题，人们很久以前就开发了许多种本地版本控制系统，大多都是采用某种简单的数据库来记录文件的历次更新差异。

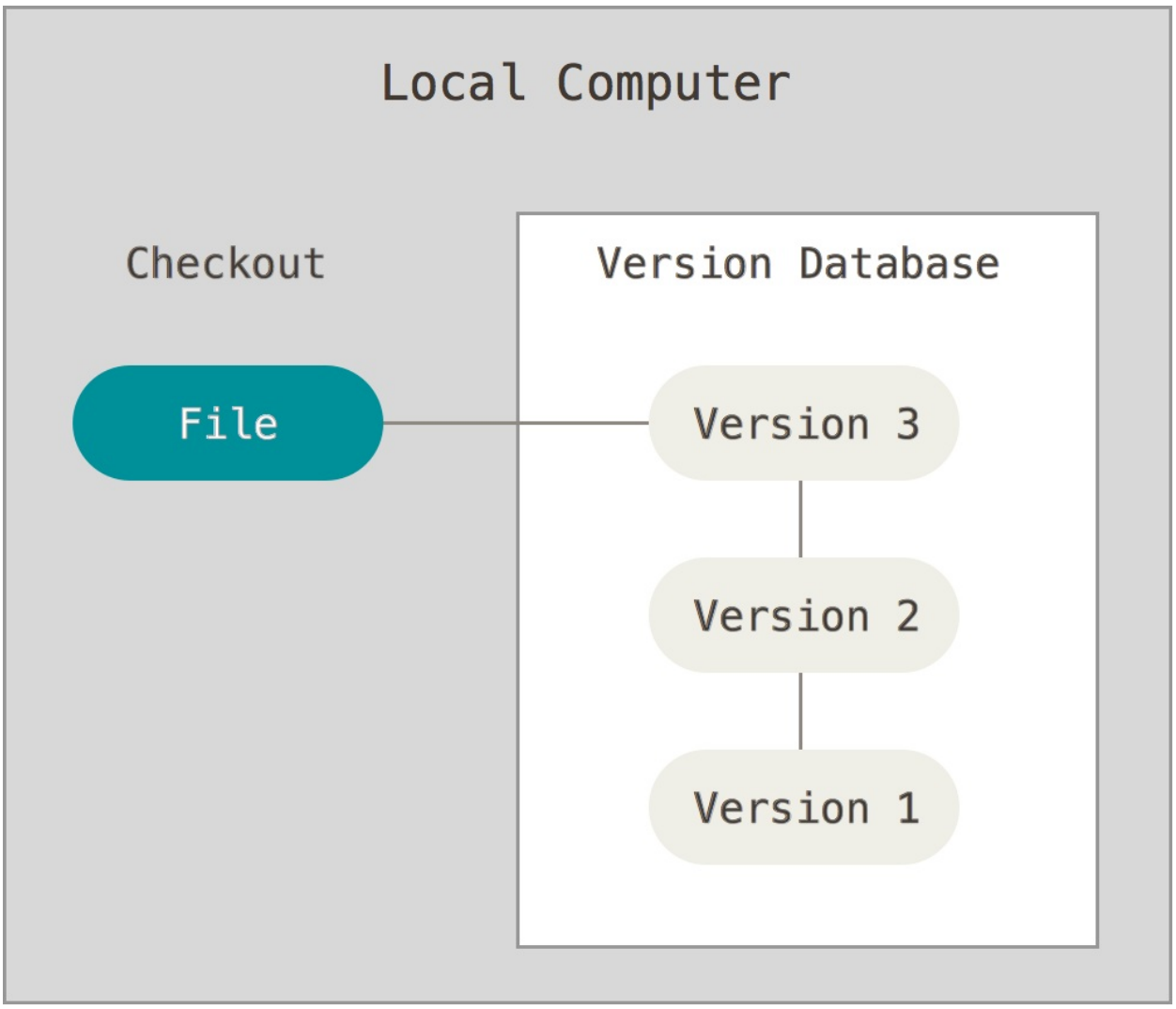


Figure 1-1. 本地版本控制.

其中最流行的一种叫做 RCS，现今许多计算机系统上都还看得到它的踪影。 甚至在流行的 Mac OS X 系统上安装了开发者工具包之后，也可以使用 `rcs` 命令。 它的工作原理是在硬盘上保存补丁集（补丁是指文件修订前后的变化）；通过应用所有的补丁，可以重新计算出各个版本的文件内容。

集中化的版本控制系统

接下来人们又遇到一个问题，如何让在不同系统上的开发者协同工作？ 于是，集中化的版本控制系统（Centralized Version Control Systems，简称 CVCS）应运而生。 这类系统，诸如 CVS、Subversion 以及 Perforce 等，都有一个单一的集中管理的服务器，保存所有文件的修订版本，而协同工作的人们都通过客户端连到这台服务器，取出最新的文件或者提交更新。 多年以来，这已成为版本控制系统的标准做法。



Figure 1-2. 集中化的版本控制.

这种做法带来了许多好处，特别是相较于老式的本地 VCS 来说。 现在，每个人都可以在一定程度上看到项目中的其他人正在做些什么。 而管理员也可以轻松掌控每个开发者的权限，并且管理一个 CVCS 要远比在各个客户端上维护本地数据库来得轻松容易。

事分两面，有好有坏。 这么做最显而易见的缺点是中央服务器的单点故障。 如果宕机一小时，那么在这一小时内，谁都无法提交更新，也就无法协同工作。 如果中心数据库所在的磁盘发生损坏，又没有做恰当备份，毫无疑问你将丢失所有数据——包括项目的整个变更历史，只剩下人们在各自机器上保留的单独快照。 本地版本控制系统也存在类似问题，只要整个项目的历史记录被保存在单一位置，就有丢失所有历史更新记录的风险。

分布式版本控制系统

于是分布式版本控制系统（Distributed Version Control System，简称 DVCS）面世了。 在这类系统中，像 Git、Mercurial、Bazaar 以及 Darcs 等，客户端并不只提取最新版本的文件快照，而是把代码仓库完整地镜像下来。 这么一来，任何一处协同工作用的服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复。 因为每一次的克隆操作，实际上都是一次对代码仓库的完整备份。

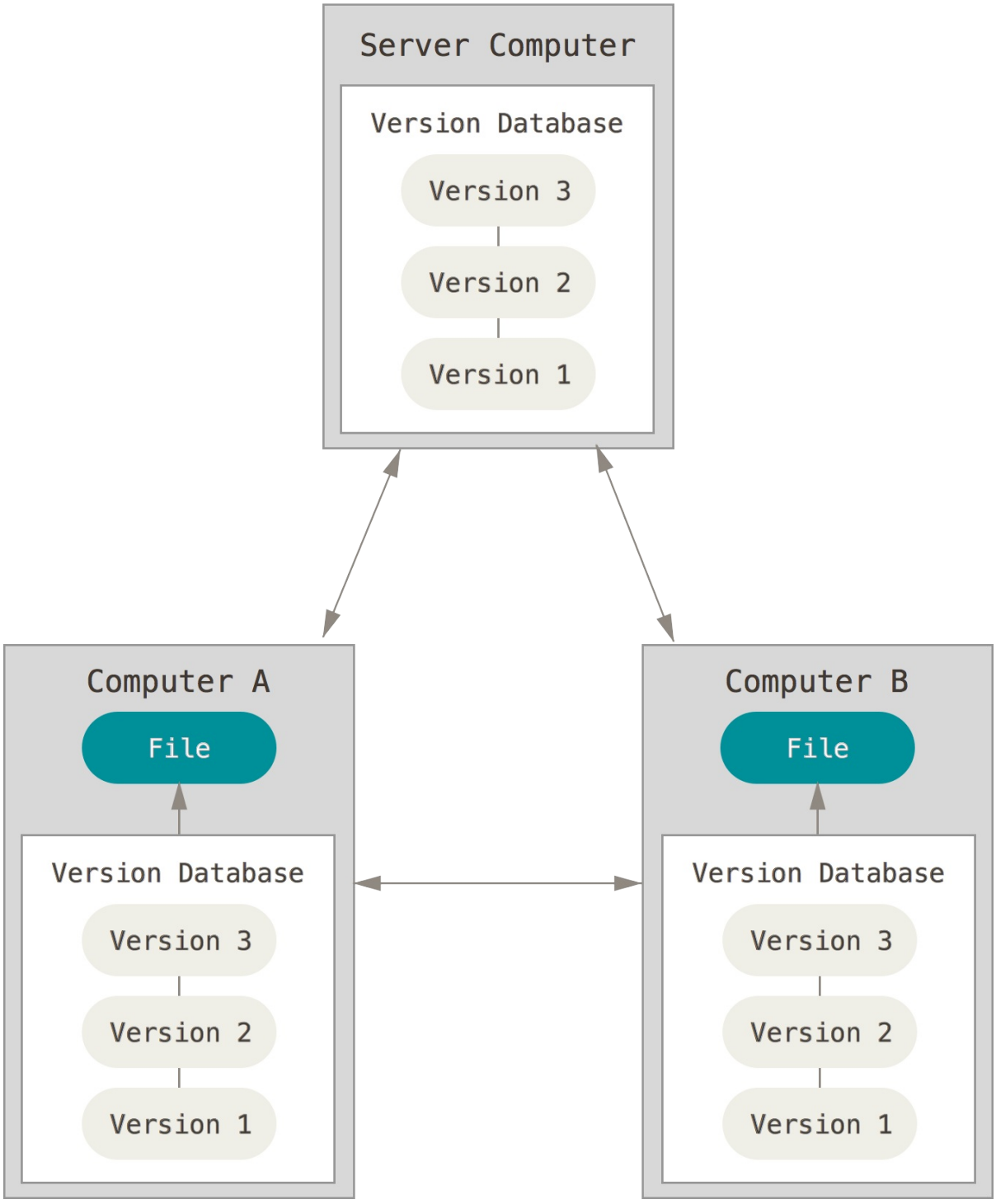


Figure 1-3. 分布式版本控制.

更进一步，许多这类系统都可以指定和若干不同的远端代码仓库进行交互。籍此，你就可以在同一个项目中，分别和不同工作小组的人相互协作。 你可以根据需要设定不同的协作流程，比如层次模型式的工作流，而这在以前的集中式系统中是无法实现的。

1.2 Git 简史

Git 简史

同生活中的许多伟大事物一样，Git 诞生于一个极富纷争大举创新的年代。

Linux 内核开源项目有着为数众广的参与者。 绝大多数的 Linux 内核维护工作都花在了提交补丁和保存归档的繁琐事务上（1991 - 2002年间）。 到 2002 年，整个项目组开始启用一个专有的分布式版本控制系统 BitKeeper 来管理和维护代码。

到了 2005 年，开发 BitKeeper 的商业公司同 Linux 内核开源社区的合作关系结束，他们收回了 Linux 内核社区免费使用 BitKeeper 的权力。 这就迫使 Linux 开源社区（特别是 Linux 的缔造者 Linux Torvalds）基于使用 BitKcheper 时的经验教训，开发出自己的版本系统。 他们对新的系统制订了若干目标：

- 速度
- 简单的设计
- 对非线性开发模式的强力支持（允许成千上万个并行开发的分支）
- 完全分布式
- 有能力高效管理类似 Linux 内核一样的超大规模项目（速度和数据量）

自诞生于 2005 年以来，Git 日臻成熟完善，在高度易用的同时，仍然保留着初期设定的目标。 它的速度飞快，极其适合管理大项目，有着令人难以置信的非线性分支管理系统（参见 Git 分支）。

1.3 Git 基础

Git 基础

那么，简单地说，Git 究竟是怎样的一个系统呢？ 请注意接下来的内容非常重要，若你理解了 Git 的思想和基本工作原理，用起来就会知其所以然，游刃有余。 在开始学习 Git 的时候，请努力分清你对其它版本管理系统的已有认识，如 Subversion 和 Perforce 等；这么做能帮助你使用工具时避免发生混淆。 Git 在保存和对待各种信息的时候与其它版本控制系统有很大差异，尽管操作起来的命令形式非常相近，理解这些差异将有助于防止你使用中的困惑。

直接记录快照，而非差异比较

Git 和其它版本控制系统（包括 Subversion 和近似工具）的主要差别在于 Git 对待数据的方法。 概念上来区分，其它大部分系统以文件变更列表的方式存储信息。 这类系统（CVS、Subversion、Perforce、Bazaar 等等）将它们保存的信息看作是一组基本文件和每个文件随时间逐步累积的差异。

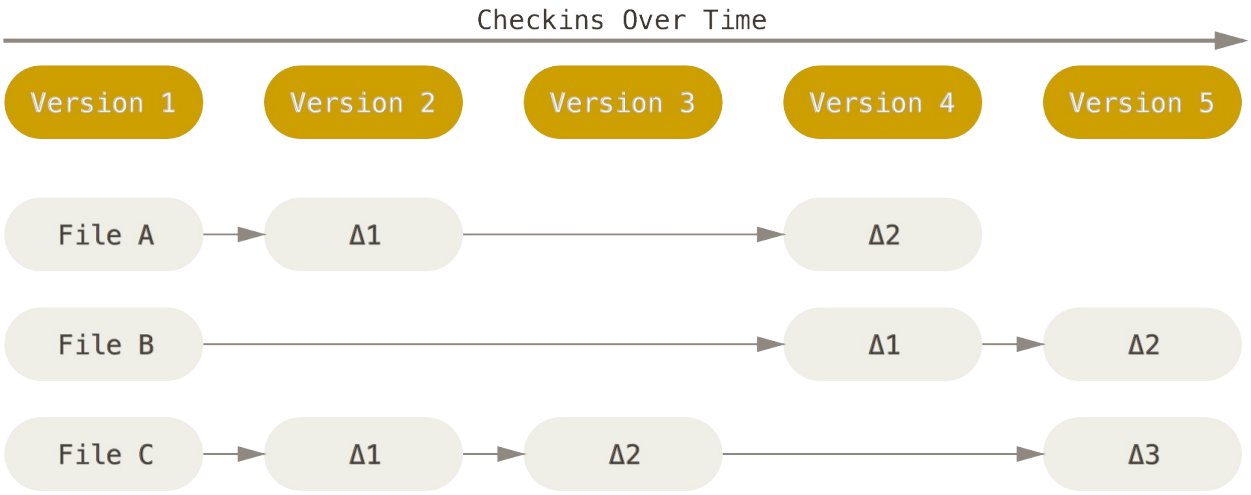


Figure 1-4. 存储每个文件与初始版本的差异.

Git 不按照以上方式对待或保存数据。 反之，Git 更像是把数据看作是对小型文件系统的一组快照。 每次你提交更新，或在 Git 中保存项目状态时，它主要对当时的全部文件制作一个快照并保存这个快照的索引。 为了高效，如果文件没有修改，Git 不再重新存储该文件，而是只保留一个链接指向之前存储的文件。 Git 对待数据更像是一个 快照流。

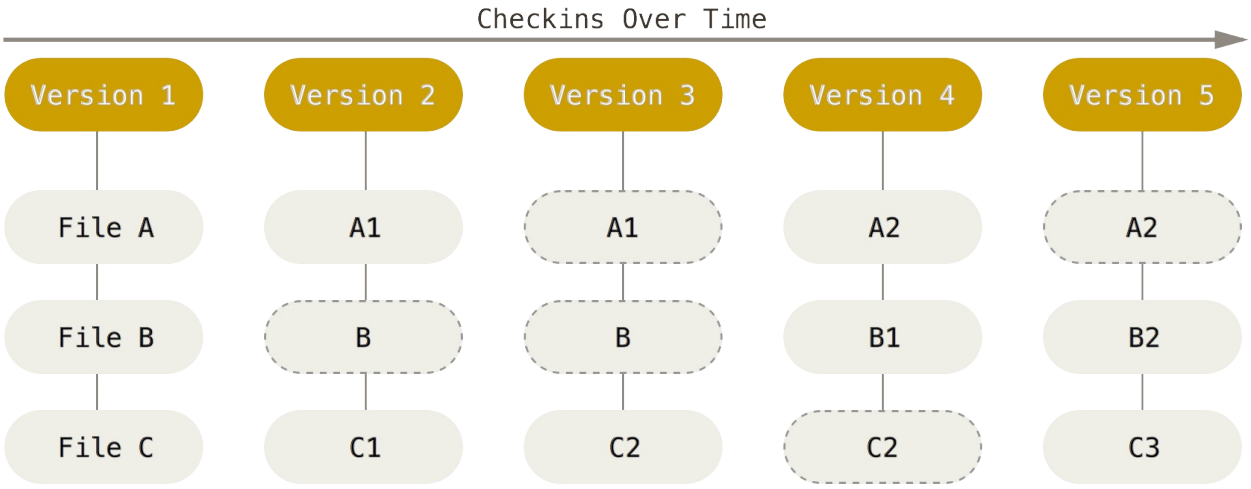


Figure 1-5. 存储项目随时间改变的快照.

这是 Git 与几乎所有其它版本控制系统的重要区别。 因此 Git 重新考虑了以前每一代版本控制系统延续下来的诸多方面。 Git 更像是一个小型的文件系统，提供了许多以此为基础构建的超强工具，而不只是一个简单的 VCS。 稍后我们在Git 分支讨论 Git 分支管理时，将探究这种方式对待数据所能获得的益处。

近乎所有操作都是本地执行

在 Git 中的绝大多数操作都只需要访问本地文件和资源，一般不需要来自网络上其它计算机的信息。 如果你习惯于所有操作都有网络延时开销的集中式版本控制系统，Git 在这方面会让你感到速度之神赐给了 Git 超凡的能量。 因为你在本地磁盘上就有项目的完整历史，所以大部分操作看起来瞬间完成。

举个例子，要浏览项目的历史，Git 不需外连到服务器去获取历史，然后再显示出来——它只需直接从本地数据库中读取。 你能立即看到项目历史。 如果你想查看当前版本与一个月前的版本之间引入的修改，Git 会查找到一个月前的文件做一次本地的差异计算，而不是由远程服务器处理或从远程服务器拉回旧版本文件再来本地处理。

这也意味着你离线或者没有 VPN 时，几乎可以进行任何操作。 如你在飞机或火车上想做些工作，你能愉快地提交，直到有网络连接时再上传。 如你回家后 VPN 客户端不正常，你仍能工作。 使用其它系统，做到如此是不可能或很费力的。 比如，用 Perforce，你没有连接服务器时几乎不能做什么事；用 Subversion 和 CVS，你能修改文件，但不能向数据库提交修改（因为你的本地数据库离线了）。 这看起来不是大问题，但是你可能会惊喜地发现它带来的巨大的不同。

Git 保证完整性

Git 中所有数据在存储前都计算校验和，然后以校验和来引用。 这意味着不可能在 Git 不知情时更改任何文件内容或目录内容。 这个功能建构在 Git 底层，是构成 Git 哲学不可或缺的部分。 若你在传送过程中丢失信息或损坏文件，Git 就能发现。

Git 用以计算校验和的机制叫做 SHA-1 散列（hash，哈希）。 这是一个由 40 个十六进制字符（0-9 和 a-f）组成字符串，基于 Git 中文件的内容或目录结构计算出来。 SHA-1 哈希看起来是这样：

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Git 中使用这种哈希值的情况很多，你将经常看到这种哈希值。 实际上，Git 数据库中保存的信息都是以文件内容的哈希值来索引，而不是文件名。

Git 一般只添加数据

你执行的 Git 操作，几乎只往 Git 数据库中增加数据。 很难让 Git 执行任何不可逆操作，或者让它以任何方式清除数据。 同别的 VCS 一样，未提交更新时有可能丢失或弄乱修改的内容；但是一旦你提交快照到 Git 中，就难以再丢失数据，特别是如果你定期的推送数据库到其它仓库的话。

这使得我们使用 Git 成为一个安心愉悦的过程，因为我们深知可以尽情做各种尝试，而没有把事情弄糟的危险。 更深度探讨 Git 如何保存数据及恢复丢失数据的话题，请参考撤消操作。

三种状态

好，请注意。 如果你希望后面的学习更顺利，记住下面这些关于 Git 的概念。 Git 有三种状态，你的文件可能处于其中之一：已提交（committed）、已修改（modified）和已暂存（staged）。 已提交表示数据已经安全的保存在本地数据库中。 已修改表示修改了文件，但还没保存到数据库中。 已暂存表示对一个已修改文件的当前版本做了标记，使之包含在下次提交的快照中。

由此引入 Git 项目的三个工作区域的概念：Git 仓库、工作目录以及暂存区域。

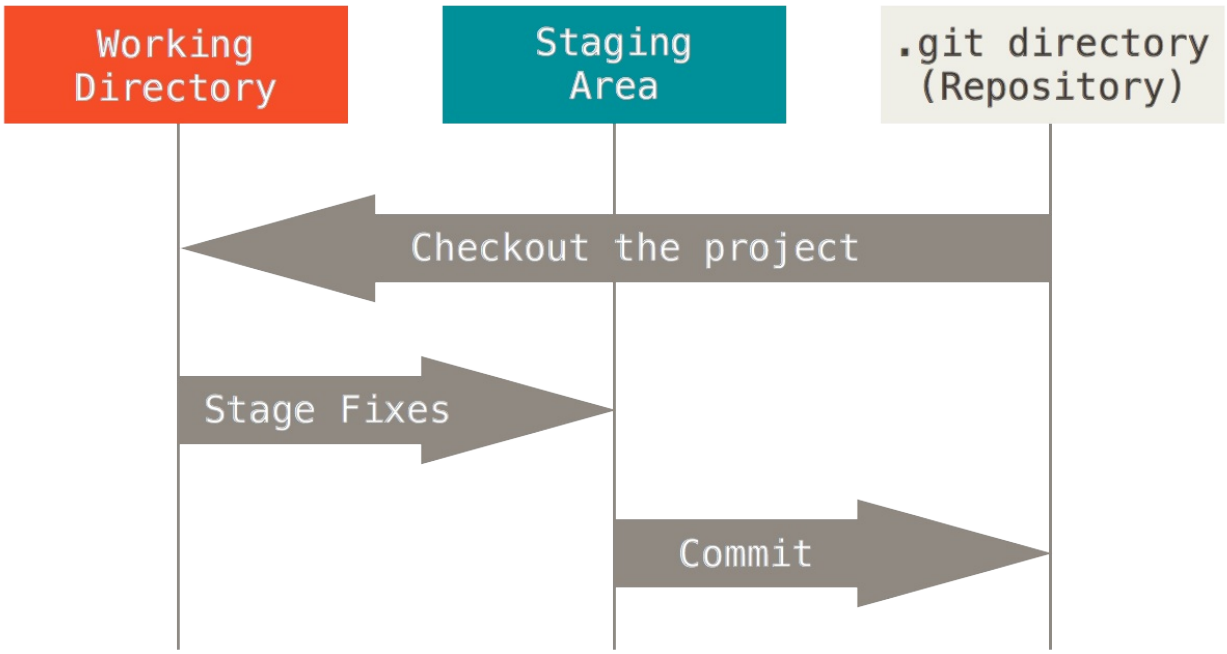


Figure 1-6. 工作目录、暂存区域以及 Git 仓库.

Git 仓库目录是 Git 用来保存项目的元数据和对象数据库的地方。 这是 Git 中最重要的部分，从其它计算机克隆仓库时，拷贝的就是这里的数据。

工作目录是对项目的某个版本独立提取出来的内容。 这些从 Git 仓库的压缩数据库中提取出来的文件，放在磁盘上供你使用或修改。

暂存区域是一个文件，保存了下次将提交的文件列表信息，一般在 Git 仓库目录中。 有时候也被称作“索引”，不过一般说法还是叫暂存区域。

基本的 Git 工作流程如下：

1. 在工作目录中修改文件。
2. 暂存文件，将文件的快照放入暂存区域。
3. 提交更新，找到暂存区域的文件，将快照永久性存储到 Git 仓库目录。

如果 Git 目录中保存着的特定版本文件，就属于已提交状态。 如果作了修改并已放入暂存区域，就属于已暂存状态。 如果自上次取出后，作了修改但还没有放到暂存区域，就是已修改状态。 在Git 基础一章，你会进一步了解这些状态的细节，并学会如何根据文件状态实施后续操作，以及怎样跳过暂存直接提交。

1.4 命令行

命令行

Git 有多种使用方式。 你可以使用原生的命令行模式，也可以使用 GUI 模式，这些 GUI 软件也能提供多种功能。 在本书中，我们将使用命令行模式。 这是因为首先，只有在命令行模式下你才能执行 Git 的 所有 命令，而大多数的 GUI 软件只实现了 Git 所有功能的一个子集以降低操作难度。 如果你学会了在命令行下如何操作，那么你在操作 GUI 软件时应该也不会遇到什么困难，但是，反之则不成立。 此外，由于每个人的想法与侧重点不同，不同的人常常会安装不同的 GUI 软件，但 所有 人一定会有命令行工具。

假如你是 Mac 用户，我们希望你懂得如何使用终端（Terminal）；假如你是 Windows 用户，我们希望你懂得如何使用命令窗口（Command Prompt）或 PowerShell。 如果你尚未掌握以上技能，我们建议你先停下来快速学习一下，本书中的讲述和举例将用到这些技能。

1.5 安装 Git

安装 Git

在你开始使用 Git 前，需要将它安装在你的计算机上。 即便已经安装，最好将它升级到最新的版本。 你可以通过软件包或者其它安装程序来安装，或者下载源码编译安装。

> NOTE
>
> 本书写作时使用的 Git 版本为 2.0.0。 我们使用的大部分命令仍然可以在很古老的 Git 版本上使用，但也有少部分命令不好用或者在旧版本中的行为有差异。 因为 Git 在保持向后兼容方便表现很好，本书使用的这些命令在 2.0 之后的版本应该有效。

在 Linux 上安装

如果你想在 Linux 上用二进制安装程序来安装 Git，可以使用发行版包含的基础软件包管理工具来安装。 如果以 Fedora 上为例，你可以使用 yum：

```
$ sudo yum install git
```

如果你在基于 Debian 的发行版上，请尝试用 apt-get：

```
$ sudo apt-get install git
```

要了解更多选择，Git 官方网站上有在各种 Unix 风格的系统上安装步骤，网址为 http://git-scm.com/download/linux。

在 Mac 上安装

在 Mac 上安装 Git 有多种方式。 最简单的方法是安装 Xcode Command Line Tools。 Mavericks（10.9）或更高版本的系统中，在 Terminal 里尝试首次运行 git 命令即可。 如果没有安装过命令行开发者工具，将会提示你安装。

如果你想安装更新的版本，可以使用二进制安装程序。 官方维护的 OSX Git 安装程序可以在 Git 官方网站下载，网址为 http://git-scm.com/download/mac。



Figure 1-7. Git OS X 安装程序.

你也可以将它作为 GitHub for Mac 的一部分来安装。 它们的图形化 Git 工具有一个安装命令行工具的选项。 你可以从 GitHub for Mac 网站下载该工具，网址为 http://mac.github.com。

在 Windows 上安装

在 Windows 上安装 Git 也有几种安装方法。 官方版本可以在 Git 官方网站下载。 打开http://git-scm.com/download/win，下载会自动开始。 要注意这是一个名为 Git for Windows的项目（也叫做 msysGit），和 Git 是分别独立的项目；更多信息请访问http://msysgit.github.io/。

另一个简单的方法是安装 GitHub for Windows。 该安装程序包含图形化和命令行版本的 Git。 它也能支持 Powershell，提供了稳定的凭证缓存和健全的 CRLF 设置。 稍后我们会对这方面有更多了解，现在只要一句话就够了，这些都是你所需要的。 你可以在 GitHub for Windows 网站下载，网址为 http://windows.github.com。

从源代码安装

有人觉得从源码安装 Git 更实用，因为你能得到最新的版本。 二进制安装程序倾向于有一些滞后，当然近几年 Git 已经成熟，这个差异不再显著。

如果你想从源码安装 Git，需要安装 Git 依赖的库：curl、zlib、openssl、expat，还有libiconv。 如果你的系统上有 yum （如 Fedora）或者 apt-get（如基于 Debian 的系统），可以使用以下命令之一来安装最小化的依赖包来编译和安装 Git 的二进制版：

```
$ sudo yum install curl-devel expat-devel gettext-devel \
openssl-devel zlib-devel
$ sudo apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
libz-dev libssl-dev
```

为了能够添加更多格式的文档（如 doc, html, info），你需要安装以下的依赖包：

```
$ sudo yum install asciidoc xmlto docbook2x
$ sudo apt-get install asciidoc xmlto docbook2x
```

当你安装好所有的必要依赖，你可以继续从几个地方来取得最新发布版本的 tar 包。 你可以从 Kernel.org 网站获取，网址为 https://www.kernel.org/pub/software/scm/git，或从 GitHub 网站上的镜像来获得，网址为 https://github.com/git/git/releases。 通常在 GitHub 上的是最新版本，但 kernel.org 上包含有文件下载签名，如果你想验证下载正确性的话会用到。

接着，编译并安装：

```
$ tar -zxf git-2.0.0.tar.gz
$ cd git-2.0.0
$ make configure
$ ./configure --prefix=/usr
$ make all doc info
$ sudo make install install-doc install-html install-info
```

完成后，你可以使用 Git 来获取 Git 的升级：

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

1.6 初次运行 Git 前的配置

初次运行 Git 前的配置

既然已经在系统上安装了 Git，你会想要做几件事来定制你的 Git 环境。 每台计算机上只需要配置一次，程序升级时会保留配置信息。 你可以在任何时候再次通过运行命令来修改它们。

Git 自带一个 `git config` 的工具来帮助设置控制 Git 外观和行为的配置变量。 这些变量存储在三个不同的位置：

1. `/etc/gitconfig` 文件: 包含系统上每一个用户及他们仓库的通用配置。 如果使用带有 `--system` 选项的 `git config` 时，它会从此文件读写配置变量。
2. `~/.gitconfig` 或 `~/.config/git/config` 文件：只针对当前用户。 可以传递 `--global` 选项让 Git 读写此文件。
3. 当前使用仓库的 Git 目录中的 `config` 文件（就是 `.git/config`）：针对该仓库。

每一个级别覆盖上一级别的配置，所以 `.git/config` 的配置变量会覆盖 `/etc/gitconfig` 中的配置变量。

在 Windows 系统中，Git 会查找 `$HOME` 目录下（一般情况下是 `C:\Users\$USER`）的 `.gitconfig` 文件。 Git 同样也会寻找 `/etc/gitconfig` 文件，但只限于 MSys 的根目录下，即安装 Git 时所选的目标位置。

用户信息

当安装完 Git 应该做的第一件事就是设置你的用户名称与邮件地址。 这样做很重要，因为每一个 Git 的提交都会使用这些信息，并且它会写入到你的每一次提交中，不可更改：

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

再次强调，如果使用了 `--global` 选项，那么该命令只需要运行一次，因为之后无论你在该系统上做任何事情， Git 都会使用那些信息。 当你想针对特定项目使用不同的用户名称与邮件地址时，可以在那个项目目录下运行没有 `--global` 选项的命令来配置。

很多 GUI 工具都会在第一次运行时帮助你配置这些信息。

文本编辑器

既然用户信息已经设置完毕，你可以配置默认文本编辑器了，当 Git 需要你输入信息时会调用它。 如果未配置，Git 会使用操作系统默认的文本编辑器，通常是 Vim。 如果你想使用不同的文本编辑器，例如

Emacs，可以这样做：

```
$ git config --global core.editor emacs
```

> WARNING
>
> Vim 和 Emacs 是像 Linux 与 Mac 等基于 Unix 的系统上开发者经常使用的流行的文本编辑器。 如果你对这些编辑器都不是很了解或者你使用的是 Windows 系统，那么可能需要搜索如何在 Git 中配置你最常用的编辑器。 如果你不设置编辑器并且不知道 Vim 或 Emacs 是什么，当它们运行起来后你可能会被弄糊涂、不知所措。

检查配置信息

如果想要检查你的配置，可以使用 `git config --list` 命令来列出所有 Git 当时能找到的配置。

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

你可能会看到重复的变量名，因为 Git 会从不同的文件中读取同一个配置（例如：`/etc/gitconfig` 与 `~/.gitconfig`）。 这种情况下，Git 会使用它找到的每一个变量的最后一个配置。

你可以通过输入 `git config <key>`： 来检查 Git 的某一项配置

```
$ git config user.name
John Doe
```

1.7 获取帮助

获取帮助

若你使用 Git 时需要获取帮助，有三种方法可以找到 Git 命令的使用手册：

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

例如，要想获得 config 命令的手册，执行

```
$ git help config
```

这些命令很棒，因为你随时随地可以使用而无需联网。 如果你觉得手册或者本书的内容还不够用，你可以尝试在 Freenode IRC 服务器（ irc.freenode.net ）的 `#git` 或 `#github` 频道寻求帮助。 这些频道经常有上百人在线，他们都精通 Git 并且乐于助人。

1.8 总结

总结

你应该已经对 Git 是什么、Git 与你可能正在使用的集中式版本控制系统有何区别等问题有了基本的了解。 现在，在你的个人系统中应该也有了一份能够工作的 Git 版本。 是时候开始学习有关 Git 的基础知识了。

2. Git 基础

第2章. Git 基础

假如你只能阅读一章来学习 Git，本章就是你的不二选择。本章内容涵盖你在使用 Git 完成各种工作中将要使用的各种基本命令。在学习完本章之后，你应该能够配置并初始化一个仓库（repository）、开始或停止跟踪（track）文件、暂存（stage）或提交（commit)更改。本章也将向你演示如何配置 Git 来忽略指定的文件和文件模式、如何迅速而简单地撤销错误操作、如何浏览你的项目的历史版本以及不同提交（commits）间的差异、如何向你的远程仓库推送（push）以及如何从你的远程仓库拉取（pull）文件。

2.1 获取 Git 仓库

获取 Git 仓库

有两种取得 Git 项目仓库的方法。第一种是在现有项目或目录下导入所有文件到 Git 中；第二种是从一个服务器克隆一个现有的 Git 仓库。

在现有目录中初始化仓库

如果你打算使用 Git 来对现有的项目进行管理，你只需要进入该项目目录并输入：

```
$ git init
```

该命令将创建一个名为 `.git` 的子目录，这个子目录含有你初始化的 Git 仓库中所有的必须文件，这些文件是 Git 仓库的骨干。但是，在这个时候，我们仅仅是做了一个初始化的操作，你的项目里的文件还没有被跟踪。(参见 Chapter 10 来了解更多关于到底 `.git` 文件夹中包含了哪些文件的信息。)

如果你是在一个已经存在文件的文件夹（而不是空文件夹）中初始化 Git 仓库来进行版本控制的话，你应该开始跟踪这些文件并提交。你可通过 `git add` 命令来实现对指定文件的跟踪，然后执行 `git commit` 提交：

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'initial project version'
```

稍后我们再逐一解释每一条指令的意思。现在，你已经得到了一个实际维护（或者说是跟踪）着若干个文件的 Git 仓库。

克隆现有的仓库

如果你想获得一份已经存在了的 Git 仓库的拷贝，比如说，你想为某个开源项目贡献自己的一份力，这时就要用到 `git clone` 命令。如果你对其它的 VCS 系统（比如说Subversion）很熟悉，请留心一下你所使用的命令是"clone"而不是"checkout"。这是 Git 区别于其它版本控制系统的一个重要特性，Git 克隆的是该 Git 仓库服务器上的几乎所有数据，而不是仅仅复制完成你的工作所需要文件。当你执行 `git clone` 命令的时候，默认配置下远程 Git 仓库中的每一个文件的每一个版本都将被拉取下来。事实上，如果你的服务器的磁盘坏掉了，你通常可以使用任何一个克隆下来的用户端来重建服务器上的仓库（虽然可能会丢失某些服务器端的挂钩设置，但是所有版本的数据仍在，详见 “在服务器上搭建 Git”）。

克隆仓库的命令格式是 `git clone [url]` 。比如，要克隆 Git 的可链接库 libgit2，可以用下面的命令：

```
$ git clone https://github.com/libgit2/libgit2
```

这会在当前目录下创建一个名为 “libgit2” 的目录，并在这个目录下初始化一个 `.git` 文件夹，从远程仓库拉取下所有数据放入 `.git` 文件夹，然后从中读取最新版本的文件的拷贝。如果你进入到这个新建的 `libgit2` 文件夹，你会发现所有的项目文件已经在里面了，准备就绪等待后续的开发和使用。如果你想在克隆远程仓库的时候，自定义本地仓库的名字，你可以使用如下命令：

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

这将执行与上一个命令相同的操作，不过在本地创建的仓库名字变为 `mylibgit`。

Git 支持多种数据传输协议。上面的例子使用的是 `https://` 协议，不过你也可以使用 `git://` 协议或者使用 SSH 传输协议，比如 `user@server:path/to/repo.git`。 “在服务器上搭建 Git” 将会介绍所有这些协议在服务器端如何配置使用，以及各种方式之间的利弊。

2.2 记录每次更新到仓库

记录每次更新到仓库

现在我们手上有了一个真实项目的 Git 仓库，并从这个仓库中取出了所有文件的工作拷贝。接下来，对这些文件做些修改，在完成了一个阶段的目标之后，提交本次更新到仓库。

请记住，你工作目录下的每一个文件都不外乎这两种状态：已跟踪或未跟踪。已跟踪的文件是指那些被纳入了版本控制的文件，在上一次快照中有它们的记录，在工作一段时间后，它们的状态可能处于未修改，已修改或已放入暂存区。工作目录中除已跟踪文件以外的所有其它文件都属于未跟踪文件，它们既不存在于上次快照的记录中，也没有放入暂存区。初次克隆某个仓库的时候，工作目录中的所有文件都属于已跟踪文件，并处于未修改状态。

编辑过某些文件之后，由于自上次提交后你对它们做了修改，Git 将它们标记为已修改文件。我们逐步将这些修改过的文件放入暂存区，然后提交所有暂存了的修改，如此反复。所以使用 Git 时文件的生命周期如下：

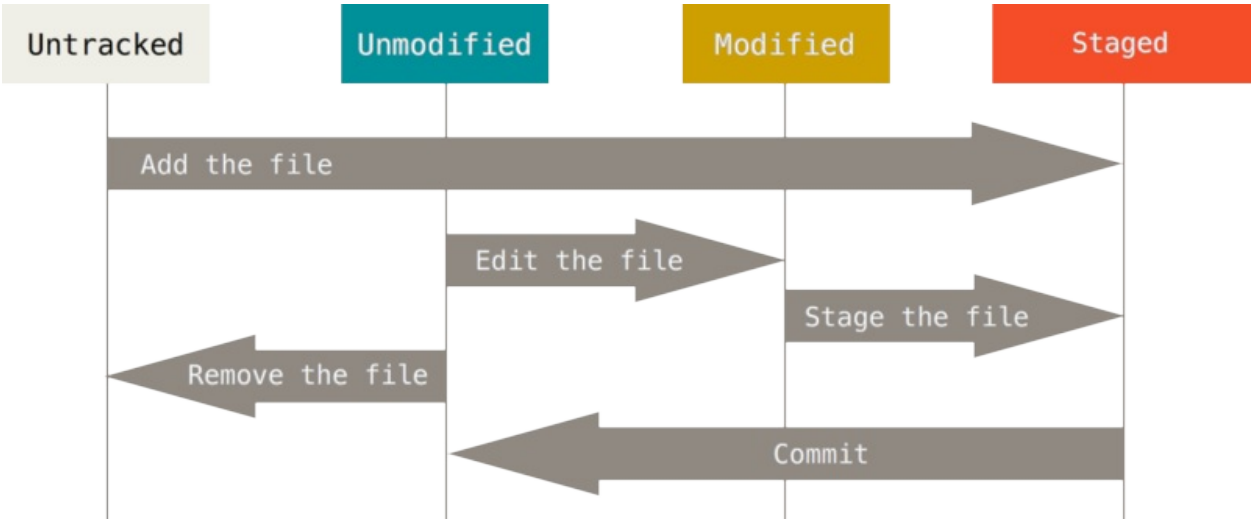


Figure 2-1. 文件的状态变化周期

检查当前文件状态

要查看哪些文件处于什么状态，可以用 `git status` 命令。如果在克隆仓库后立即使用此命令，会看到类似这样的输出：

```
$ git status
On branch master
nothing to commit, working directory clean
```

这说明你现在的工作目录相当干净。换句话说，所有已跟踪文件在上次提交后都未被更改过。此外，上面

的信息还表明，当前目录下没有出现任何处于未跟踪状态的新文件，否则 Git 会在这里列出来。最后，该命令还显示了当前所在分支，并告诉你这个分支同远程服务器上对应的分支没有偏离。现在，分支名是“master”,这是默认的分支名。我们在 Chapter 3 会详细讨论分支和引用。

现在，让我们在项目下创建一个新的 README 文件。如果之前并不存在这个文件，使用 `git status` 命令，你将看到一个新的未跟踪文件：

```
$ echo 'My Project' > README
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README

nothing added to commit but untracked files present (use "git add" to track)
```

在状态报告中可以看到新建的 README 文件出现在 `Untracked files` 下面。未跟踪的文件意味着 Git 在之前的快照（提交）中没有这些文件；Git 不会自动将之纳入跟踪范围，除非你明明白白地告诉它“我需要跟踪该文件”，这样的处理让你不必担心将生成的二进制文件或其它不想被跟踪的文件包含进来。不过现在的例子中，我们确实想要跟踪管理 README 这个文件。

跟踪新文件

使用命令 `git add` 开始跟踪一个文件。所以，要跟踪 README 文件，运行：

```
$ git add README
```

此时再运行 `git status` 命令，会看到 README 文件已被跟踪，并处于暂存状态：

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
```

只要在 `Changes to be committed` 这行下面的，就说明是已暂存状态。如果此时提交，那么该文件此时此刻的版本将被留存在历史记录中。你可能会想起之前我们使用 `git init` 后就运行了 `git add (files)` 命令，开始跟踪当前目录下的文件。 `git add` 命令使用文件或目录的路径作为参数；如果参数是目录的路径，该命令将递归地跟踪该目录下的所有文件。

暂存已修改文件

现在我们来修改一个已被跟踪的文件。如果你修改了一个名为 `CONTRIBUTING.md` 的已被跟踪的文件，然后运行 `git status` 命令，会看到下面内容：

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

文件 `CONTRIBUTING.md` 出现在 `Changes not staged for commit` 这行下面，说明已跟踪文件的内容发生了变化，但还没有放到暂存区。要暂存这次更新，需要运行 `git add` 命令。这是个多功能命令：可以用它开始跟踪新文件，或者把已跟踪的文件放到暂存区，还能用于合并时把有冲突的文件标记为已解决状态等。将这个命令理解为“添加内容到下一次提交中”而不是“将一个文件添加到项目中”要更加合适。现在让我们运行 `git add` 将"CONTRIBUTING.md"放到暂存区，然后再看看 `git status` 的输出：

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

现在两个文件都已暂存，下次提交时就会一并记录到仓库。假设此时，你想要在 `CONTRIBUTING.md` 里再加条注释，重新编辑存盘后，准备好提交。不过且慢，再运行 `git status` 看看：

```
$ vim CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

怎么回事？现在 `CONTRIBUTING.md` 文件同时出现在暂存区和非暂存区。这怎么可能呢？好吧，实际上 Git 只不过暂存了你运行 `git add` 命令时的版本，如果你现在提交， `CONTRIBUTING.md` 的版本是你最后一次运行 `git add` 命令时的那个版本，而不是你运行 `git commit` 时，在工作目录中的当前版

本。所以，运行了 `git add` 之后又作了修订的文件，需要重新运行 `git add` 把最新版本重新暂存起来：

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

状态简览

`git status` 命令的输出十分详细，但其用语有些繁琐。如果你使用 `git status -s` 命令或 `git status --short` 命令，你将得到一种更为紧凑的格式输出。运行 `git status -s` ，状态报告输出如下：

```
$ git status -s
 M README
MM Rakefile
A  lib/git.rb
M  lib/simplegit.rb
?? LICENSE.txt
```

新添加的未跟踪文件前面有 `??` 标记，新添加到暂存区中的文件前面有 `A` 标记，修改过的文件前面有 `M` 标记。你可能注意到了 `M` 有两个可以出现的位置，出现在右边的 `M` 表示该文件被修改了但是还没放入暂存区，出现在靠左边的 `M` 表示该文件被修改了并放入了暂存区。例如，上面的状态报告显示：`README` 文件在工作区被修改了但是还没有将修改后的文件放入暂存区，`lib/simplegit.rb` 文件被修改了并将修改后的文件放入了暂存区。而 `Rakefile` 在工作区被修改并提交到暂存区后又在工作区中被修改了，所以在暂存区和工作区都有该文件被修改了的记录。

忽略文件

一般我们总会有些文件无需纳入 Git 的管理，也不希望它们总出现在未跟踪文件列表。通常都是些自动生成的文件，比如日志文件，或者编译过程中创建的临时文件等。在这种情况下，我们可以创建一个名为 `.gitignore` 的文件，列出要忽略的文件模式。来看一个实际的例子：

```
$ cat .gitignore
*.[oa]
*~
```

第一行告诉 Git 忽略所有以 `.o` 或 `.a` 结尾的文件。一般这类对象文件和存档文件都是编译过程中出现的。第二行告诉 Git 忽略所有以波浪符（~）结尾的文件，许多文本编辑软件（比如 Emacs）都用这样的文件名保存副本。此外，你可能还需要忽略 log，tmp 或者 pid 目录，以及自动生成的文档等等。要养成一开始就设置好 .gitignore 文件的习惯，以免将来误提交这类无用的文件。

文件 `.gitignore` 的格式规范如下：

- 所有空行或者以 `#` 开头的行都会被 Git 忽略。
- 可以使用标准的 glob 模式匹配。
- 匹配模式可以以（ `/` ）开头防止递归。
- 匹配模式可以以（ `/` ）结尾指定目录。
- 要忽略指定模式以外的文件或目录，可以在模式前加上惊叹号（ `!` ）取反。

所谓的 glob 模式是指 shell 所使用的简化了的正则表达式。星号（ `*` ）匹配零个或多个任意字符；`[abc]` 匹配任何一个列在方括号中的字符（这个例子要么匹配一个 a，要么匹配一个 b，要么匹配一个 c）；问号（ `?` ）只匹配一个任意字符；如果在方括号中使用短划线分隔两个字符，表示所有在这两个字符范围内的都可以匹配（比如 `[0-9]` 表示匹配所有 0 到 9 的数字）。使用两个星号（ `*` ）表示匹配任意中间目录，比如 `a/**/z` 可以匹配 `a/z`，`a/b/z` 或 `a/b/c/z` 等。

我们再看一个 .gitignore 文件的例子：

```
# no .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO
/TODO

# ignore all files in the build/ directory
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .pdf files in the doc/ directory
doc/**/*.pdf
```

GitHub 有一个十分详细的针对数十种项目及语言的 `.gitignore` 文件列表，你可以在 https://github.com/github/gitignore 找到它.

查看已暂存和未暂存的修改

如果 `git status` 命令的输出对于你来说过于模糊，你想知道具体修改了什么地方，可以用 `git diff` 命令。稍后我们会详细介绍 `git diff`，你可能通常会用它来回答这两个问题：当前做的哪些更新还没有暂存？有哪些更新已经暂存起来准备好了下次提交？尽管 `git status` 已经通过在相应栏下列出文件名的方式回答了这个问题，`git diff` 将通过文件补丁的格式显示具体哪些行发生了改变。

假如再次修改 README 文件后暂存，然后编辑 `CONTRIBUTING.md` 文件后先不暂存，运行 `status` 命令将会看到：

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

要查看尚未暂存的文件更新了哪些部分，不加参数直接输入 `git diff`：

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

此命令比较的是工作目录中当前文件和暂存区域快照之间的差异，也就是修改之后还没有暂存起来的变化内容。

若要查看已暂存的将要添加到下次提交里的内容，可以用 `git diff --cached` 命令。（Git 1.6.1 及更高版本还允许使用 `git diff --staged`，效果是相同的，但更好记些。）

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

请注意，git diff 本身只显示尚未暂存的改动，而不是自上次提交以来所做的所有改动。所以有时候你一下子暂存了所有更新过的文件后，运行 `git diff` 后却什么也没有，就是这个原因。

像之前说的，暂存 `CONTRIBUTING.md` 后再编辑，运行 `git status` 会看到暂存前后的两个版本。如果我们的环境（终端输出）看起来如下：

```
$ git add CONTRIBUTING.md
$ echo '# test line' >> CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

现在运行 `git diff` 看暂存前后的变化：

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 643e24f..87f08c8 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -119,3 +119,4 @@ at the
 ## Starter Projects

 See our [projects list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).
+# test line
```

然后用 `git diff --cached` 查看已经暂存起来的变化：（--staged 和 --cached 是同义词）

```
$ git diff --cached
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

> Git Diff 的插件版本

> 在本书中，我们使用 `git diff` 来分析文件差异。但是，如果你喜欢通过图形化的方式或其它格式

> 输出方式的话，可以使用 `git difftool` 命令来用 Araxis ，emerge 或 vimdiff 等软件输出 diff 分析结果。使用 `git difftool --tool-help` 命令来看你的系统支持哪些 Git Diff 插件。

提交更新

现在的暂存区域已经准备妥当可以提交了。在此之前，请一定要确认还有什么修改过的或新建的文件还没有 `git add` 过，否则提交的时候不会记录这些还没暂存起来的变化。这些修改过的文件只保留在本地磁盘。所以，每次准备提交前，先用 `git status` 看下，是不是都已暂存起来了，然后再运行提交命令 `git commit` ：

```
$ git commit
```

这种方式会启动文本编辑器以便输入本次提交的说明。(默认会启用 shell 的环境变量 `$EDITOR` 所指定的软件，一般都是 vim 或 emacs。当然也可以按照 Chapter 1 介绍的方式，使用 `git config --global core.editor` 命令设定你喜欢的编辑软件。)

编辑器会显示类似下面的文本信息（本例选用 Vim 的屏显方式展示）：

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#       new file:   README
#       modified:   CONTRIBUTING.md
#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

可以看到，默认的提交消息包含最后一次运行 `git status` 的输出，放在注释行里，另外开头还有一空行，供你输入提交说明。你完全可以去掉这些注释行，不过留着也没关系，多少能帮你回想起这次更新的内容有哪些。(如果想要更详细的对修改了哪些内容的提示，可以用 `-v` 选项，这会将你所做的改变的 diff 输出放到编辑器中从而使你知道本次提交具体做了哪些修改。) 退出编辑器时，Git 会丢掉注释行，用你输入提交附带信息生成一次提交。

另外，你也可以在 `commit` 命令后添加 `-m` 选项，将提交信息与命令放在同一行，如下所示：

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 README
```

好，现在你已经创建了第一个提交！可以看到，提交后它会告诉你，当前是在哪个分支（ `master` ）提交的，本次提交的完整 SHA-1 校验和是什么（ `463dc4f` ），以及在本次提交中，有多少文件修订过，多

少行添加和删改过。

请记住，提交时记录的是放在暂存区域的快照。任何还未暂存的仍然保持已修改状态，可以在下次提交时纳入版本管理。每一次运行提交操作，都是对你项目作一次快照，以后可以回到这个状态，或者进行比较。

跳过使用暂存区域

尽管使用暂存区域的方式可以精心准备要提交的细节，但有时候这么做略显繁琐。Git 提供了一个跳过使用暂存区域的方式，只要在提交的时候，给 `git commit` 加上 `-a` 选项，Git 就会自动把所有已经跟踪过的文件暂存起来一并提交，从而跳过 `git add` 步骤：

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 file changed, 5 insertions(+), 0 deletions(-)
```

看到了吗？提交之前不再需要 `git add` 文件“CONTRIBUTING.md”了。

移除文件

要从 Git 中移除某个文件，就必须要从已跟踪文件清单中移除（确切地说，是从暂存区域移除），然后提交。可以用 `git rm` 命令完成此项工作，并连带从工作目录中删除指定的文件，这样以后就不会出现在未跟踪文件清单中了。

如果只是简单地从工作目录中手工删除文件，运行 `git status` 时就会在 “Changes not staged for commit” 部分（也就是 未暂存清单）看到：

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    PROJECTS.md

no changes added to commit (use "git add" and/or "git commit -a")
```

然后再运行 `git rm` 记录此次移除文件的操作：

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    PROJECTS.md
```

下一次提交时，该文件就不再纳入版本管理了。如果删除之前修改过并且已经放到暂存区域的话，则必须要用强制删除选项 `-f`（译注：即 force 的首字母）。这是一种安全特性，用于防止误删还没有添加到快照的数据，这样的数据不能被 Git 恢复。

另外一种情况是，我们想把文件从 Git 仓库中删除（亦即从暂存区域移除），但仍然希望保留在当前工作目录中。换句话说，你想让文件保留在磁盘，但是并不想让 Git 继续跟踪。当你忘记添加 `.gitignore` 文件，不小心把一个很大的日志文件或一堆 `.a` 这样的编译生成文件添加到暂存区时，这一做法尤其有用。为达到这一目的，使用 `--cached` 选项：

```
$ git rm --cached README
```

`git rm` 命令后面可以列出文件或者目录的名字，也可以使用 `glob` 模式。比方说：

```
$ git rm log/\*.log
```

注意到星号 `*` 之前的反斜杠 `\`，因为 Git 有它自己的文件模式扩展匹配方式，所以我们不用 shell 来帮忙展开。此命令删除 `log/` 目录下扩展名为 `.log` 的所有文件。类似的比如：

```
$ git rm \*~
```

该命令为删除以 `~` 结尾的所有文件。

移动文件

不像其它的 VCS 系统，Git 并不显式跟踪文件移动操作。如果在 Git 中重命名了某个文件，仓库中存储的元数据并不会体现出这是一次改名操作。不过 Git 非常聪明，它会推断出究竟发生了什么，至于具体是如何做到的，我们稍后再谈。

既然如此，当你看到 Git 的 `mv` 命令时一定会困惑不已。要在 Git 中对文件改名，可以这么做：

```
$ git mv file_from file_to
```

它会恰如预期般正常工作。实际上，即便此时查看状态信息，也会明白无误地看到关于重命名操作的说

明：

```
$ git mv README.md README
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

其实，运行 `git mv` 就相当于运行了下面三条命令：

```
$ mv README.md README
$ git rm README.md
$ git add README
```

如此分开操作，Git 也会意识到这是一次改名，所以不管何种方式结果都一样。两者唯一的区别是，`mv` 是一条命令而另一种方式需要三条命令，直接用 `git mv` 轻便得多。不过有时候用其他工具批处理改名的话，要记得在提交前删除老的文件名，再添加新的文件名。

2.3 查看提交历史

查看提交历史

在提交了若干更新，又或者克隆了某个项目之后，你也许想回顾下提交历史.完成这个任务最简单而又有效的工具是 `git log` 命令。

接下来的例子会用我专门用于演示的 simplegit 项目，运行下面的命令获取该项目源代码：

```
git clone https://github.com/schacon/simplegit-progit
```

然后在此项目中运行 git log，应该会看到下面的输出：

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

默认不用任何参数的话，`git log` 会按提交时间列出所有的更新，最近的更新排在最上面。正如你所看到的，这个命令会列出每个提交的 SHA-1 校验和、作者的名字和电子邮件地址、提交时间以及提交说明。

`git log` 有许多选项可以帮助你搜寻你所要找的提交，接下来我们介绍些最常用的。

一个常用的选项是 `-p`，用来显示每次提交的内容差异。你也可以加上 `-2` 来仅显示最近两次提交：

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
```

```
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.platform  =   Gem::Platform::RUBY
     s.name      =   "simplegit"
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gee-mail.com"
     s.summary   =   "A simple gem for using Git in Ruby code."

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
     end

 end
-
-if $0 == __FILE__
-  git = SimpleGit.new
-  puts git.show
-end
\ No newline at end of file
```

该选项除了显示基本信息之外，还在附带了每次 commit 的变化。当进行代码审查，或者快速浏览某个搭档提交的 commit 所带来的变化的时候，这个参数就非常有用了。你也可以为 `git log` 附带一系列的总结性选项。比如说，如果你想看到每次提交的简略的统计信息，你可以使用 `--stat` 选项：

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

 Rakefile | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

 lib/simplegit.rb | 5 -----
 1 file changed, 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700
```

```
    first commit

 README           |  6 ++++++
 Rakefile         | 23 +++++++++++++++++++++
 lib/simplegit.rb | 25 +++++++++++++++++++++++++
 3 files changed, 54 insertions(+)
```

正如你所看到的，`--stat` 选项在每次提交的下面列出额所有被修改过的文件、有多少文件被修改了以及被修改过的文件的哪些行被移除或是添加了。在每次提交的最后还有一个总结。

另外一个常用的选项是 `--pretty`。这个选项可以指定使用不同于默认格式的方式展示提交历史。这个选项有一些内建的子选项供你使用。比如用 `oneline` 将每个提交放在一行显示，查看的提交数很大时非常有用。另外还有 `short`，`full` 和 `fuller` 可以用，展示的信息或多或少有些不同，请自己动手实践一下看看效果如何。

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

但最有意思的是 format，可以定制要显示的记录格式。这样的输出对后期提取分析格外有用 — 因为你知道输出的格式不会随着Git的更新而发生改变：

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : changed the version number
085bb3b - Scott Chacon, 6 years ago : removed unnecessary test
a11bef0 - Scott Chacon, 6 years ago : first commit
```

Table 2-1 列出了常用的格式占位符写法及其代表的意义。

Table 2-1. `git log --pretty=format` 常用的选项

选项	说明
`%H`	提交对象（commit）的完整哈希字串
`%h`	提交对象的简短哈希字串
`%T`	树对象（tree）的完整哈希字串
`%t`	树对象的简短哈希字串
`%P`	父对象（parent）的完整哈希字串
`%p`	父对象的简短哈希字串
`%an`	作者（author）的名字
`%ae`	作者的电子邮件地址
`%ad`	作者修订日期（可以用 --date= 选项定制格式）
`%ar`	作者修订日期，按多久以前的方式显示

`%cn`	提交者(committer)的名字
`%ce`	提交者的电子邮件地址
`%cd`	提交日期
`%cr`	提交日期，按多久以前的方式显示
`%s`	提交说明

你一定奇怪 作者 和 提交者 之间究竟有何差别，其实作者指的是实际作出修改的人，提交者指的是最后将此工作成果提交到仓库的人。所以，当你为某个项目发布补丁，然后某个核心成员将你的补丁并入项目时，你就是作者，而那个核心成员就是提交者。我们会在 Chapter 5 再详细介绍两者之间的细微差别。

当 oneline 或 format 与另一个 `log` 选项 `--graph` 结合使用时尤其有用。这个选项添加了一些ASCII字符串来形象地展示你的分支、合并历史：

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
*  5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
|/
* d6016bc require time for xmlschema
*  11d191e Merge branch 'defunkt' into local
```

这种输出类型会在我们下一张学完分支与合并以后变得更加有趣。

以上只是简单介绍了一些 `git log` 命令支持的选项。Table 2-2 列出了我们目前涉及到的和没涉及到的选项，已经它们是如何影响 log 命令的输出的：

Table 2-2. `git log` 的常用选项

选项	说明
`-p`	按补丁格式显示每个更新之间的差异。
`--stat`	显示每次更新的文件修改统计信息。
`--shortstat`	只显示 --stat 中最后的行数修改添加移除统计。
`--name-only`	仅在提交信息后显示已修改的文件清单。
`--name-status`	显示新增、修改、删除的文件清单。
`--abbrev-commit`	仅显示 SHA-1 的前几个字符，而非所有的 40 个字符。
`--relative-date`	使用较短的相对时间显示（比如，“2 weeks ago”）。
`--graph`	显示 ASCII 图形表示的分支合并历史。
`--pretty`	使用其他格式显示历史提交信息。可用的选项包括 oneline，short，full，fuller 和 format（后跟指定格式）。

限制输出长度

除了定制输出格式的选项之外，`git log` 还有许多非常实用的限制输出长度的选项，也就是只输出部分提交信息。之前你已经看到过 `-2` 了，它只显示最近的两条提交，实际上，这是 `-<n>` 选项的写法，其中的 `n` 可以是任何整数，表示仅显示最近的若干条提交。不过实践中我们是不太用这个选项的，Git 在输出所有提交时会自动调用分页程序，所以你一次只会看到一页的内容。

另外还有按照时间作限制的选项，比如 `--since` 和 `--until` 也很有用。例如，下面的命令列出所有最近两周内的提交：

```
$ git log --since=2.weeks
```

这个命令可以在多种格式下工作，比如说具体的某一天 `"2008-01-15"`，或者是相对地多久以前 `"2 years 1 day 3 minutes ago"`。

还可以给出若干搜索条件，列出符合的提交。用 `--author` 选项显示指定作者的提交，用 `--grep` 选项搜索提交说明中的关键字。（请注意，如果要得到同时满足这两个选项搜索条件的提交，就必须用 `--all-match` 选项。否则，满足任意一个条件的提交都会被匹配出来）

另一个非常有用的筛选选项是 `-S`，可以列出那些添加或移除了某些字符串的提交。比如说，你想找出添加或移除了某一个特定函数的引用的提交，你可以这样使用：

```
$ git log -Sfunction_name
```

最后一个很实用的 `git log` 选项是路径(path)，如果只关心某些文件或者目录的历史提交，可以在 git log 选项的最后指定它们的路径。因为是放在最后位置上的选项，所以用两个短划线（--）隔开之前的选项和后面限定的路径名。

在 Table 2-3 中列出了常用的选项

Table 2-3. 限制 `git log` 输出的选项

选项	说明
`-(n)`	仅显示最近的 n 条提交
`--since`, `--after`	仅显示指定时间之后的提交。
`--until`, `--before`	仅显示指定时间之前的提交。
`--author`	仅显示指定作者相关的提交。
`--committer`	仅显示指定提交者相关的提交。
`--grep`	仅显示含指定关键字的提交
`-S`	仅显示添加或移除了某个关键字的提交

来看一个实际的例子，如果要查看 Git 仓库中，2008 年 10 月期间，Junio Hamano 提交的但未合并的测试文件，可以用下面的查询命令：

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
   --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attributes are in use
acd3b9e - Enhance hold_lock_file_for_{update,append}() API
f563754 - demonstrate breakage of detached checkout with symbolic link HEAD
d1a43f2 - reset --hard/read-tree --reset -u: remove unmerged new paths
51a94af - Fix "checkout --track -b newbranch" on detached HEAD
b0ad11e - pull: allow "git pull origin $something:$current_branch" into an unborn branch
```

在近40000条提交中，上面的输出仅列出了符合条件的6条记录。

2.4 撤消操作

撤消操作

在任何一个阶段，你都有可能想要撤消某些操作。这里，我们将会学习几个撤消你所做修改的基本工具。注意，有些撤消操作是不可逆的。这是在使用 Git 的过程中，会因为操作失误而导致之前的工作丢失的少有的几个地方之一。

有时候我们提交完了才发现漏掉了几个文件没有添加，或者提交信息写错了。此时，可以运行带有 `--amend` 选项的提交命令尝试重新提交：

```
$ git commit --amend
```

这个命令会将暂存区中的文件提交。如果自上次提交以来你还未做任何修改（例如，在上次提交后马上执行了此命令），那么快照会保持不变，而你所修改的只是提交信息。

文本编辑器启动后，可以看到之前的提交信息。编辑后保存会覆盖原来的提交信息。

例如，你提交后发现忘记了暂存某些需要的修改，可以像下面这样操作：

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

最终你只会有一个提交 - 第二次提交将代替第一次提交的结果。

取消暂存的文件

接下来的两个小节演示如何操作暂存区域与工作目录中已修改的文件。这些命令在修改文件状态的同时，也会提示如何撤消操作。例如，你已经修改了两个文件并且想要将它们作为两次独立的修改提交，但是却意外地输入了 `git add *` 暂存了它们两个。如何只取消暂存两个中的一个呢？`git status` 命令提示了你：

```
$ git add *
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
    modified:   CONTRIBUTING.md
```

在 “Changes to be committed” 文字正下方，提示使用 `git reset HEAD <file>...` 来取消暂

存。所以，我们可以这样来取消暂存 `CONTRIBUTING.md` 文件：

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
M       CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

这个命令有点儿奇怪，但是起作用了。`CONTRIBUTING.md` 文件已经是修改未暂存的状态了。

虽然在调用时加上 `--hard` 选项可以令 `git reset` 成为一个危险的命令（译注：可能导致工作目录中所有当前进度丢失！），但本例中工作目录内的文件并不会被修改。不加选项地调用 `git reset` 并不危险 — 它只会修改暂存区域。

到目前为止这个神奇的调用就是你需要对 `git reset` 命令了解的全部。我们将会在 “重置揭密” 中了解 `reset` 的更多细节以及如何掌握它做一些真正有趣的事。

撤消对文件的修改

如果你并不想保留对 `CONTRIBUTING.md` 文件的修改怎么办？你该如何方便地撤消修改 - 将它还原成上次提交时的样子（或者刚克隆完的样子，或者刚把它放入工作目录时的样子）？幸运的是，`git status` 也告诉了你应该如何做。在最后一个例子中，未暂存区域是这样：

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

它非常清楚地告诉了你如何撤消之前所做的修改。让我们来按照提示执行：

```
$ git checkout -- CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

可以看到那些修改已经被撤消了。

你需要知道 `git checkout -- [file]` 是一个危险的命令，这很重要。你对那个文件做的任何修改都会消失 - 你只是拷贝了另一个文件来覆盖它。除非你确实清楚不想要那个文件了，否则不要使用这个命令。

如果你仍然想保留对那个文件做出的修改，但是现在仍然需要撤消，我们将会在 Chapter 3 介绍保存进度与分支；这些通常是更好的做法。

记住，在 Git 中任何 已提交的 东西几乎总是可以恢复的。甚至那些被删除的分支中的提交或使用 `--amend` 选项覆盖的提交也可以恢复（阅读 “数据恢复” 了解数据恢复）。然而，任何你未提交的东西丢失后很可能再也找不到了。

2.5 远程仓库的使用

远程仓库的使用

为了能在任意 Git 项目上协作，你需要知道如何管理自己的远程仓库。远程仓库是指托管在因特网或其他网络中的你的项目的版本库。你可以有好几个远程仓库，通常有些仓库对你只读，有些则可以读写。与他人协作涉及管理远程仓库以及根据需要推送或拉取数据。管理远程仓库包括了解如何添加远程仓库、移除无效的远程仓库、管理不同的远程分支并定义它们是否被跟踪等等。在本节中，我们将介绍一部分远程管理的技能。

查看远程仓库

如果想查看你已经配置的远程仓库服务器，可以运行 `git remote` 命令。它会列出你指定的每一个远程服务器的简写。如果你已经克隆了自己的仓库，那么至少应该能看到 origin - 这是 Git 给你克隆的仓库服务器的默认名字：

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

你也可以指定选项 `-v`，会显示需要读写远程仓库使用的 Git 保存的简写与其对应的 URL。

```
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
```

如果你的远程仓库不止一个，该命令会将它们全部列出。例如，与几个协作者合作的，拥有多个远程仓库的仓库看起来像下面这样：

```
$ cd grit
$ git remote -v
bakkdoor  https://github.com/bakkdoor/grit (fetch)
bakkdoor  https://github.com/bakkdoor/grit (push)
cho45     https://github.com/cho45/grit (fetch)
cho45     https://github.com/cho45/grit (push)
defunkt   https://github.com/defunkt/grit (fetch)
defunkt   https://github.com/defunkt/grit (push)
koke      git://github.com/koke/grit.git (fetch)
koke      git://github.com/koke/grit.git (push)
origin    git@github.com:mojombo/grit.git (fetch)
```

```
origin    git@github.com:mojombo/grit.git (push)
```

这样我们可以轻松拉取其中任何一个用户的贡献。此外，我们大概还会有某些远程仓库的推送权限，虽然我们目前还不会在此介绍。

注意这些远程仓库使用了不同的协议；我们将会在 “在服务器上搭建 Git” 中了解关于它们的更多信息。

添加远程仓库

我在之前的章节中已经提到并展示了如何添加远程仓库的示例，不过这里将告诉你如何明确地做到这一点。运行 `git remote add <shortname> <url>` 添加一个新的远程 Git 仓库，同时指定一个你可以轻松引用的简写：

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
pb      https://github.com/paulboone/ticgit (fetch)
pb      https://github.com/paulboone/ticgit (push)
```

现在你可以在命令行中使用字符串 `pb` 来代替整个 URL。例如，如果你想拉取 Paul 的仓库中有但你没有的信息，可以运行 `git fetch pb`：

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

现在 Paul 的 master 分支可以在本地通过 `pb/master` 访问到 - 你可以将它合并到自己的某个分支中，或者如果你想要查看它的话，可以检出一个指向该点的本地分支。（我们将会在 Chapter 3 中详细介绍什么是分支以及如何使用分支。）

从远程仓库中抓取与拉取

就如刚才所见，从远程仓库中获得数据，可以执行：

```
$ git fetch [remote-name]
```

这个命令会访问远程仓库，从中拉取所有你还没有的数据。执行完成后，你将会拥有那个远程仓库中所有分支的引用，可以随时合并或查看。

如果你使用 `clone` 命令克隆了一个仓库，命令会自动将其添加为远程仓库并默认以 “origin” 为简写。所以，`git fetch origin` 会抓取克隆（或上一次抓取）后新推送的所有工作。必须注意 `git fetch` 命令会将数据拉取到你的本地仓库 - 它并不会自动合并或修改你当前的工作。当准备好时你必须手动将其合并入你的工作。

如果你有一个分支设置为跟踪一个远程分支（阅读下一节与 Chapter 3 了解更多信息），可以使用 `git pull` 命令来自动的抓取然后合并远程分支到当前分支。这对你来说可能是一个更简单或更舒服的工作流程；默认情况下，`git clone` 命令会自动设置本地 master 分支跟踪克隆的远程仓库的 master 分支（或不管是什么名字的默认分支）。运行 `git pull` 通常会从最初克隆的服务器上抓取数据并自动尝试合并到当前所在的分支。

推送到远程仓库

当你想分享你的项目时，必须将其推送到上游。这个命令很简单：`git push [remote-name] [branch-name]`。当你想要将 master 分支推送到 `origin` 服务器时（再次说明，克隆时通常会自动帮你设置好那两个名字），那么运行这个命令就可以将你所做的备份到服务器：

```
$ git push origin master
```

只有当你有所克隆服务器的写入权限，并且之前没有人推送过时，这条命令才能生效。当你和其他人在同一时间克隆，他们先推送到上游然后你再推送到上游，你的推送就会毫无疑问地被拒绝。你必须先将他们的工作拉取下来并将其合并进你的工作后才能推送。阅读 Chapter 3 了解如何推送到远程仓库服务器的详细信息。

查看远程仓库

如果想要查看某一个远程仓库的更多信息，可以使用 `git remote show [remote-name]` 命令。如果想以一个特定的缩写名运行这个命令，例如 `origin`，会得到像下面类似的信息：

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
  Remote branches:
    master                               tracked
    dev-branch                           tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

它同样会列出远程仓库的 URL 与跟踪分支的信息。这些信息非常有用，它告诉你正处于 master 分支，并且如果运行 git pull，就会抓取所有的远程引用，然后将远程 master 分支合并到本地 master 分支。它也会列出拉取到的所有远程引用。

这是一个经常遇到的简单例子。如果你是 Git 的重度使用者，那么还可以通过 `git remote show` 看到更多的信息。

```
$ git remote show origin
* remote origin
  URL: https://github.com/my-org/complex-project
  Fetch URL: https://github.com/my-org/complex-project
  Push  URL: https://github.com/my-org/complex-project
  HEAD branch: master
  Remote branches:
    master                           tracked
    dev-branch                       tracked
    markdown-strip                   tracked
    issue-43                         new (next fetch will store in remotes/origin)
    issue-45                         new (next fetch will store in remotes/origin)
    refs/remotes/origin/issue-11     stale (use 'git remote prune' to remove)
  Local branches configured for 'git pull':
    dev-branch merges with remote dev-branch
    master     merges with remote master
  Local refs configured for 'git push':
    dev-branch                     pushes to dev-branch                     (up to date)
    markdown-strip                 pushes to markdown-strip                 (up to date)
    master                         pushes to master                         (up to date)
```

这个命令列出了当你在特定的分支上执行 `git push` 会自动地推送到哪一个远程分支。它也同样地列出了哪些远程分支不在你的本地，哪些远程分支已经从服务器上移除了，还有当你执行 `git pull` 时哪些分支会自动合并。

远程仓库的移除与重命名

如果想要重命名引用的名字可以运行 `git remote rename` 去修改一个远程仓库的简写名。例如，想要将 `pb` 重命名为 `paul`，可以用 `git remote rename` 这样做：

```
$ git remote rename pb paul
$ git remote
origin
paul
```

值得注意的是这同样也会修改你的远程分支名字。那些过去引用 `pb/master` 的现在会引用 `paul/master`。

如果因为一些原因想要移除一个远程仓库 - 你已经从服务器上搬走了或不再想使用某一个特定的镜像了，又或者某一个贡献者不再贡献了 - 可以使用 `git remote rm` ：

```
$ git remote rm paul
$ git remote
origin
```

2.6 打标签

打标签

像其他版本控制系统（VCS）一样，Git 可以给历史中的某一个提交打上标签，以示重要。比较有代表性的是人们会使用这个功能来标记发布结点（v1.0 等等）。在本节中，你将会学习如何列出已有的标签、如何创建新标签、以及不同类型的标签分别是什么。

列出标签

在 Git 中列出已有的标签是非常简单直观的。只需要输入 `git tag`：

```
$ git tag
v0.1
v1.3
```

这个命令以字母顺序列出标签；但是它们出现的顺序并不重要。

你也可以使用特定的模式查找标签。例如，Git 自身的源代码仓库包含标签的数量超过 500 个。如果只对 1.8.5 系列感兴趣，可以运行：

```
$ git tag -l 'v1.8.5*'
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

创建标签

Git 使用两种主要类型的标签：轻量标签（lightweight）与附注标签（annotated）。

一个轻量标签很像一个不会改变的分支 - 它只是一个特定提交的引用。

然而，附注标签是存储在 Git 数据库中的一个完整对象。它们是可以被校验的；其中包含打标签者的名字、电子邮件地址、日期时间；还有一个标签信息；并且可以使用 GNU Privacy Guard（GPG）签名与验证。通常建议创建附注标签，这样你可以拥有以上所有信息；但是如果你只是想用一个临时的标签，或者因为某些原因不想要保存那些信息，轻量标签也是可用的。

附注标签

在 Git 中创建一个附注标签是很简单的。最简单的方式是当你在运行 `tag` 命令时指定 `-a` 选项：

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
```

`-m` 选项指定了一条将会存储在标签中的信息。如果没有为附注标签指定一条信息，Git 会运行编辑器要求你输入信息。

通过使用 `git show` 命令可以看到标签信息与对应的提交信息：

```
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

输出显示了打标签者的信息、打标签的日期时间、附注信息，然后显示具体的提交信息。

轻量标签

另一种给提交打标签的方式是使用轻量标签。轻量标签本质上是将提交校验和存储到一个文件中 - 没有保存任何其他信息。创建轻量标签，不需要使用 `-a`、`-s` 或 `-m` 选项，只需要提供标签名字：

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

这时，如果在标签上运行 `git show`，你不会看到额外的标签信息。命令只会显示出提交信息：

```
$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

后期打标签

你也可以对过去的提交打标签。假设提交历史是这样的：

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

现在，假设在 v1.2 时你忘记给项目打标签，也就是在 “updated rakefile” 提交。你可以在之后补上标签。要在那个提交上打标签，你需要在命令的末尾指定提交的校验和（或部分校验和）：

```
$ git tag -a v1.2 9fceb02
```

可以看到你已经在那次提交上打上标签了：

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

    updated rakefile
...
```

共享标签

默认情况下，`git push` 命令并不会传送标签到远程仓库服务器上。在创建完标签后你必须显式地推送标签到共享服务器上。这个过程就像共享远程分支一样 - 你可以运行 `git push origin [tagname]`。

```
$ git push origin v1.5
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
```

```
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.5 -> v1.5
```

如果想要一次性推送很多标签，也可以使用带有 `--tags` 选项的 `git push` 命令。这将会把所有不在远程仓库服务器上的标签全部传送到那里。

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.4 -> v1.4
 * [new tag]         v1.4-lw -> v1.4-lw
```

现在，当其他人从仓库中克隆或拉取，他们也能得到你的那些标签。

检出标签

在 Git 中你并不能真的检出一个标签，因为它们并不能像分支一样来回移动。如果你想要工作目录与仓库中特定的标签版本完全一样，可以使用 `git checkout -b [branchname] [tagname]` 在特定的标签上创建一个新分支：

```
$ git checkout -b version2 v2.0.0
Switched to a new branch 'version2'
```

当然，如果在这之后又进行了一次提交，`version2` 分支会因为改动向前移动了，那么 `version2` 分支就会和 `v2.0.0` 标签稍微有些不同，这时就应该当心了。

2.7 Git 别名

Git 别名

在我们结束本章 Git 基础之前，正好有一个小技巧可以使你的 Git 体验更简单、容易、熟悉：别名。我们不会在之后的章节中引用到或假定你使用过它们，但是你大概应该知道如何使用它们。

Git 并不会在你输入部分命令时自动推断出你想要的命令。如果不想每次都输入完整的 Git 命令，可以通过 `git config` 文件来轻松地为每一个命令设置一个别名。这里有一些例子你可以试试：

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

这意味着，当要输入 `git commit` 时，只需要输入 `git ci`。随着你继续不断地使用 Git，可能也会经常使用其他命令，所以创建别名时不要犹豫。

在创建你认为应该存在的命令时这个技术会很有用。例如，为了解决取消暂存文件的易用性问题，可以向 Git 中添加你自己的取消暂存别名：

```
$ git config --global alias.unstage 'reset HEAD --'
```

这会使下面的两个命令等价：

```
$ git unstage fileA
$ git reset HEAD -- fileA
```

这样看起来更清楚一些。通常也会添加一个 `last` 命令，像这样：

```
$ git config --global alias.last 'log -1 HEAD'
```

这样，可以轻松地看到最后一次提交：

```
$ git last
commit 66938dae3329c7aebe598c2246a8e6af90d04646
Author: Josh Goebel <dreamer3@example.com>
Date:   Tue Aug 26 19:48:51 2008 +0800

    test for current head

    Signed-off-by: Scott Chacon <schacon@example.com>
```

可以看出，Git 只是简单地将别名替换为对应的命令。然而，你可能想要执行外部命令，而不是一个 Git 子命令。如果是那样的话，可以在命令前面加入 `!` 符号。如果你自己要写一些与 Git 仓库协作的工具的话，那会很有用。我们现在演示将 `git visual` 定义为 `gitk` 的别名：

```
$ git config --global alias.visual '!gitk'
```

2.8 总结

总结

现在，你可以完成所有基本的 Git 本地操作 - 创建或者克隆一个仓库、做更改、暂存并提交这些更改、浏览你的仓库从创建到现在的所有更改的历史。下一步，本书将介绍 Git 的杀手级特性：分支模型。

3. Git 分支

第3章. Git 分支

几乎所有的版本控制系统都以某种形式支持分支。使用分支意味着你可以把你的工作从开发主线上分离开来，以免影响开发主线。在很多版本控制系统中，这是一个略微低效的过程——常常需要完全创建一个源代码目录的副本。对于大项目来说，这样的过程会耗费很多时间。

有人把 Git 的分支模型称为它的“必杀技特性”，也正因为这一特性，使得 Git 从众多版本控制系统中脱颖而出。为何 Git 的分支模型如此出众呢？Git 处理分支的方式可谓是难以置信的轻量，创建新分支这一操作几乎能在瞬间完成，并且在不同分支之间的切换操作也是一样便捷。与许多其它版本控制系统不同，Git 鼓励在工作流程中频繁地使用分支与合并，哪怕一天之内进行许多次。理解和精通这一特性，你便会意识到 Git 是如此的强大而又独特，并且从此真正改变你的开发方式。

3.1 分支简介

分支简介

为了真正理解 Git 处理分支的方式，我们需要回顾一下 Git 是如何保存数据的。

或许你还记得 Chapter 1 的内容，Git 保存的不是文件的变化或者差异，而是一系列不同时刻的文件快照。

在进行提交操作时，Git 会保存一个提交对象（commit object）。知道了 Git 保存数据的方式，我们可以很自然的想到——该提交对象会包含一个指向暂存内容快照的指针。但不仅仅是这样，该提交对象还包含了作者的姓名和邮箱、提交时输入的信息以及指向它的父对象的指针。首次提交产生的提交对象没有父对象，普通提交操作产生的提交对象有一个父对象，而由多个分支合并产生的提交对象有多个父对象，

为了说得更加形象，我们假设现在有一个工作目录，里面包含了三个将要被暂存和提交的文件。暂存操作会为每一个文件计算校验和（使用我们在 Chapter 1 中提到的 SHA-1 哈希算法），然后会把当前版本的文件快照保存到 Git 仓库中（Git 使用 blob 对象来保存它们），最终将校验和加入到暂存区域等待提交：

```
$ git add README test.rb LICENSE
$ git commit -m 'The initial commit of my project'
```

当使用 `git commit` 进行提交操作时，Git 会先计算每一个子目录（本例中只有项目根目录）的校验和，然后在 Git 仓库中这些校验和保存为树对象。随后，Git 便会创建一个提交对象，它除了包含上面提到的那些信息外，还包含指向这个树对象（项目根目录）的指针。如此一来，Git 就可以在需要的时候重现此次保存的快照。

现在，Git 仓库中有五个对象：三个 blob 对象（保存着文件快照）、一个树对象（记录着目录结构和 blob 对象索引）以及一个提交对象（包含着指向前述树对象的指针和所有提交信息）。

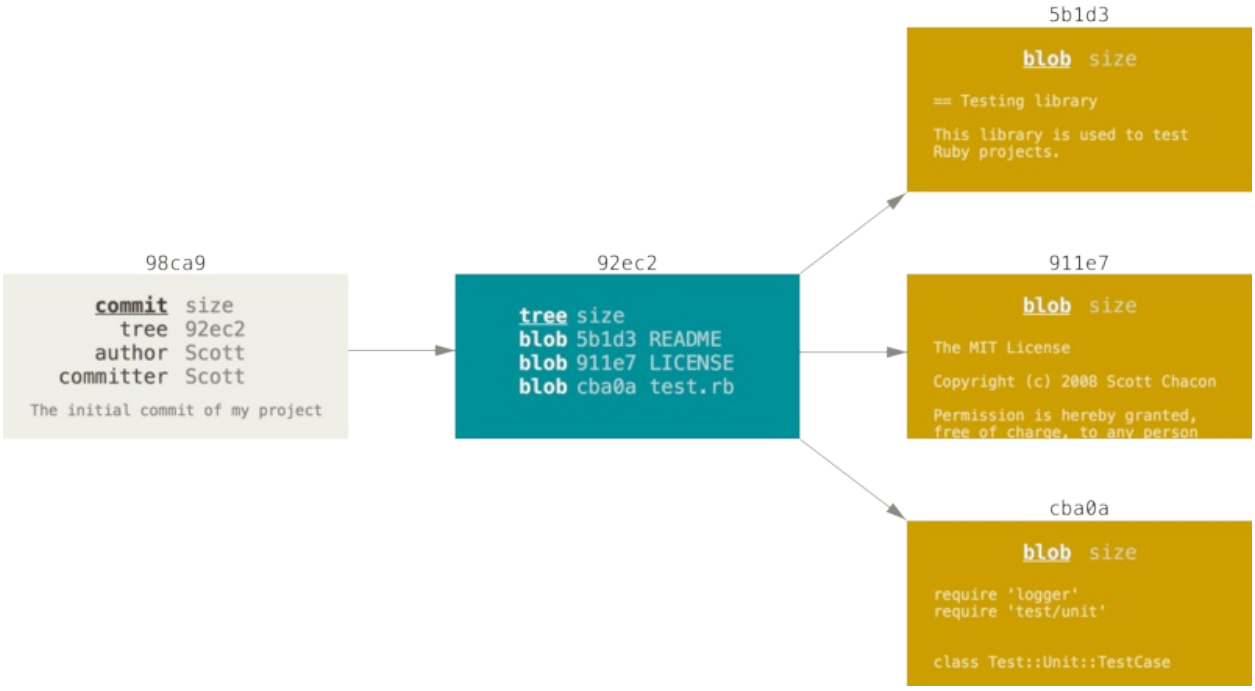


Figure 3-1. 首次提交对象及其树结构

做些修改后再次提交，那么这次产生的提交对象会包含一个指向上次提交对象（父对象）的指针。

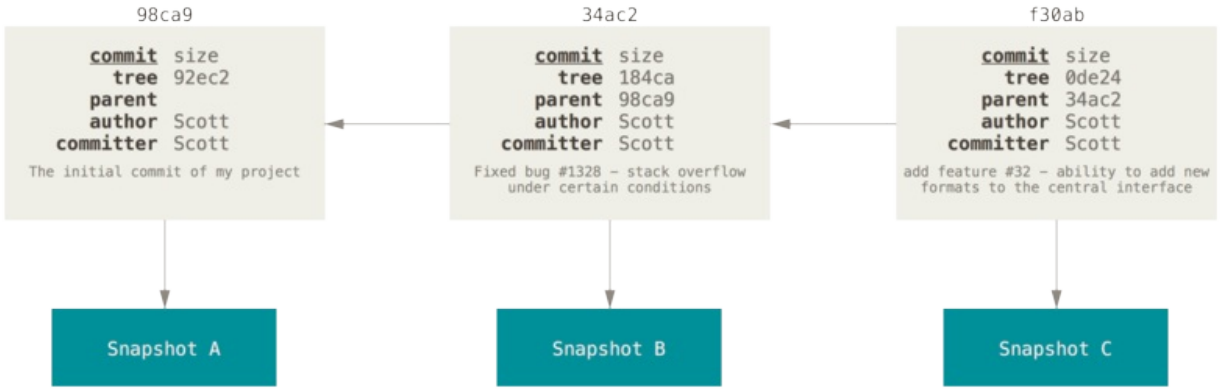


Figure 3-2. 提交对象及其父对象

Git 的分支，其实本质上仅仅是指向提交对象的可变指针。Git 的默认分支名字是 `master`。在多次提交操作之后，你其实已经有一个指向最后那个提交对象的 `master` 分支。它会在每次的提交操作中自动向前移动。

Git 的 “master” 分支并不是一个特殊分支。它就跟其它分支完全没有区别。之所以几乎每一个仓库都有 master 分支，是因为 `git init` 命令默认创建它，并且大多数人都懒得去改动它。

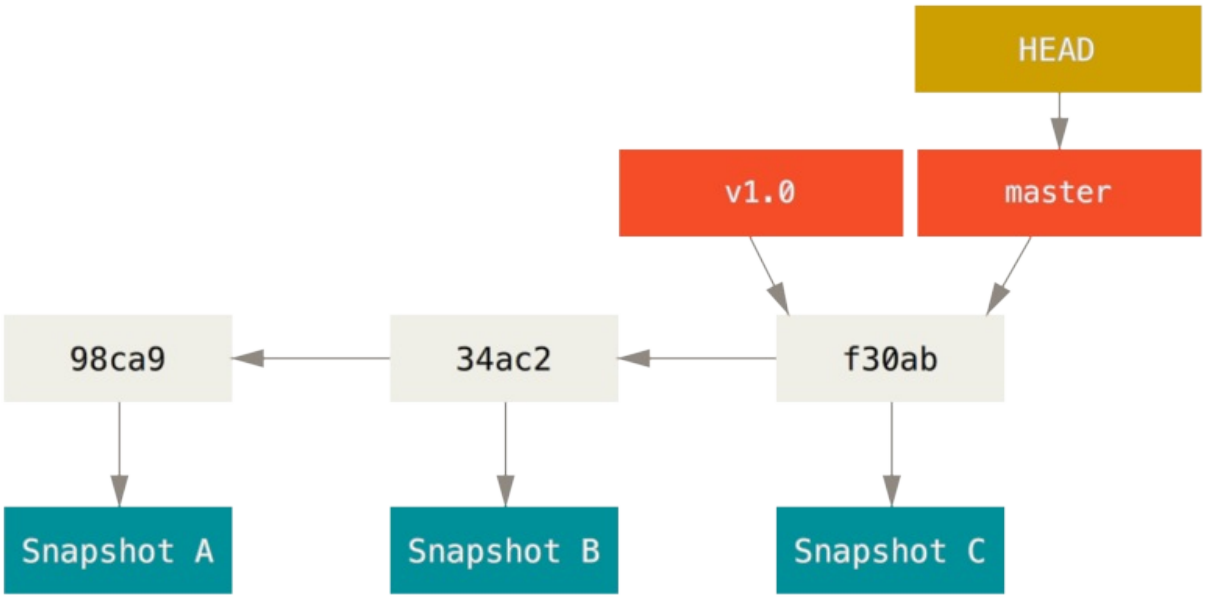


Figure 3-3. 分支及其提交历史

分支创建

Git 是怎么创建新分支的呢？很简单，它只是为你创建了一个可以移动的新的指针。比如，创建一个 testing 分支，你需要使用 `git branch` 命令：

```
$ git branch testing
```

这会在当前所在的提交对象上创建一个指针。

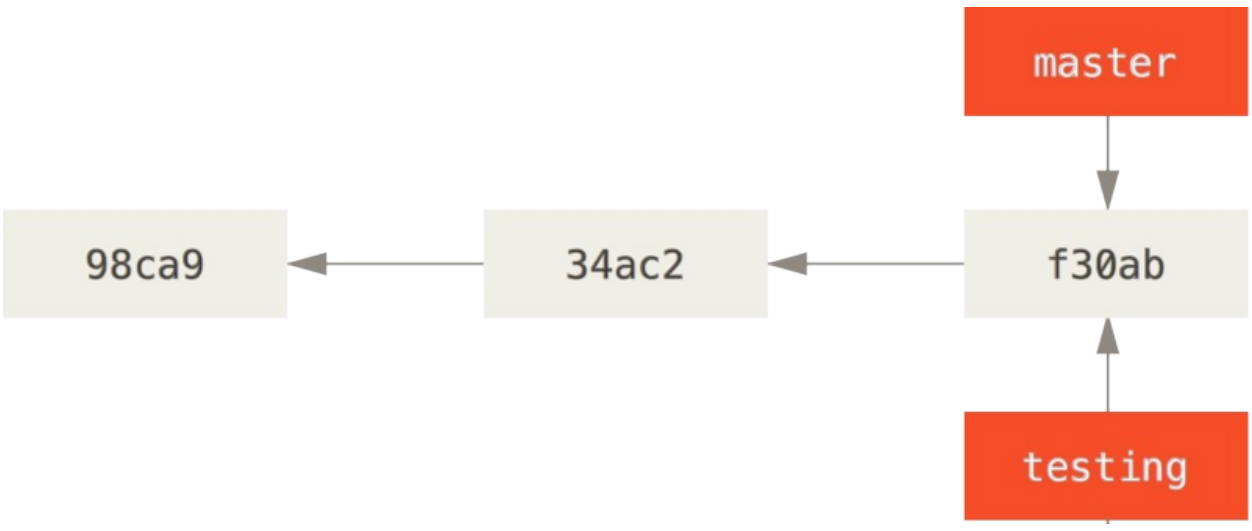


Figure 3-4. 两个指向相同提交历史的分支

那么，Git 又是怎么知道当前在哪一个分支上呢？也很简单，它有一个名为 `HEAD` 的特殊指针。请注意它和许多其它版本控制系统（如 Subversion 或 CVS）里的 `HEAD` 概念完全不同。在 Git 中，它是一个指针，指向当前所在的本地分支（译注：将 `HEAD` 想象为当前分支的别名）。在本例中，你仍然在 `master` 分支上。因为 `git branch` 命令仅仅 创建 一个新分支，并不会自动切换到新分支中去。

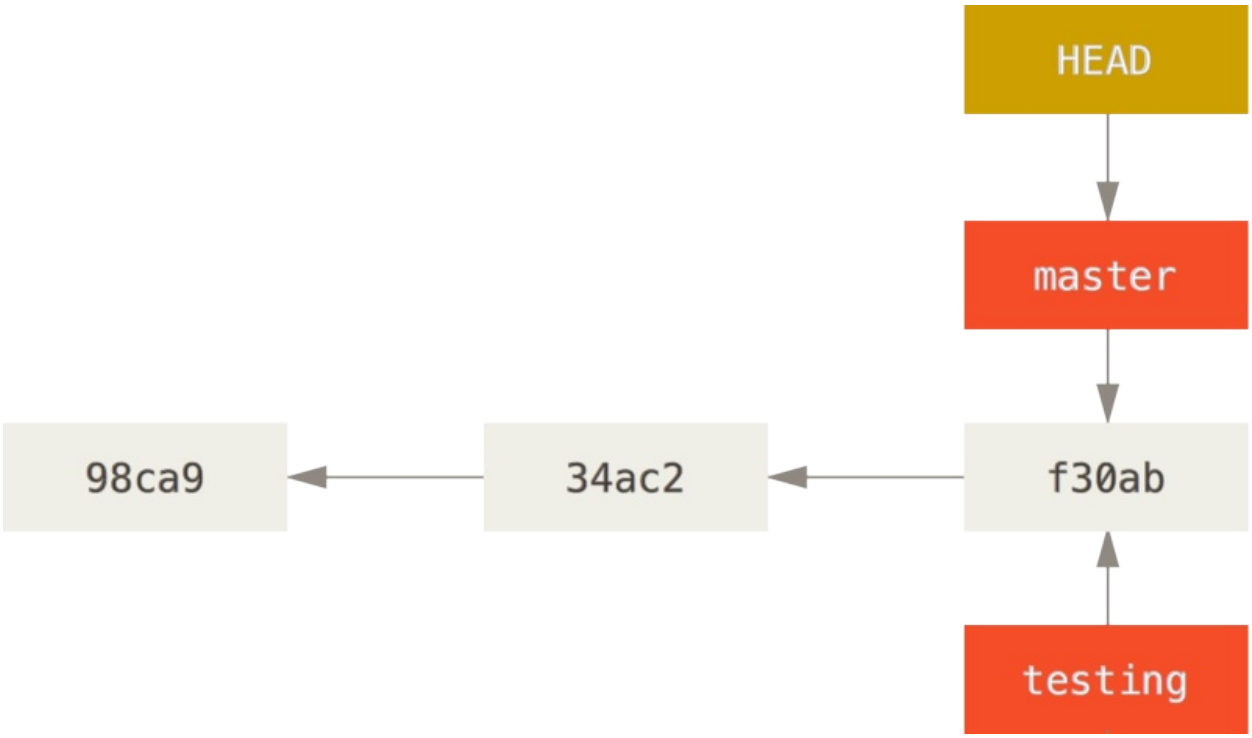


Figure 3-5. HEAD 指向当前所在的分支

你可以简单地使用 `git log` 命令查看各个分支当前所指的对象。提供这一功能的参数是 `--decorate`。

```
$ git log --oneline --decorate
f30ab (HEAD, master, testing) add feature #32 - ability to add new
34ac2 fixed bug #1328 - stack overflow under certain conditions
98ca9 initial commit of my project
```

正如你所见，当前 “master” 和 “testing” 分支均指向校验和以 `f30ab` 开头的提交对象。

分支切换

要切换到一个已存在的分支，你需要使用 `git checkout` 命令。我们现在切换到新创建的 `testing` 分支去：

```
$ git checkout testing
```

这样 `HEAD` 就指向 `testing` 分支了。

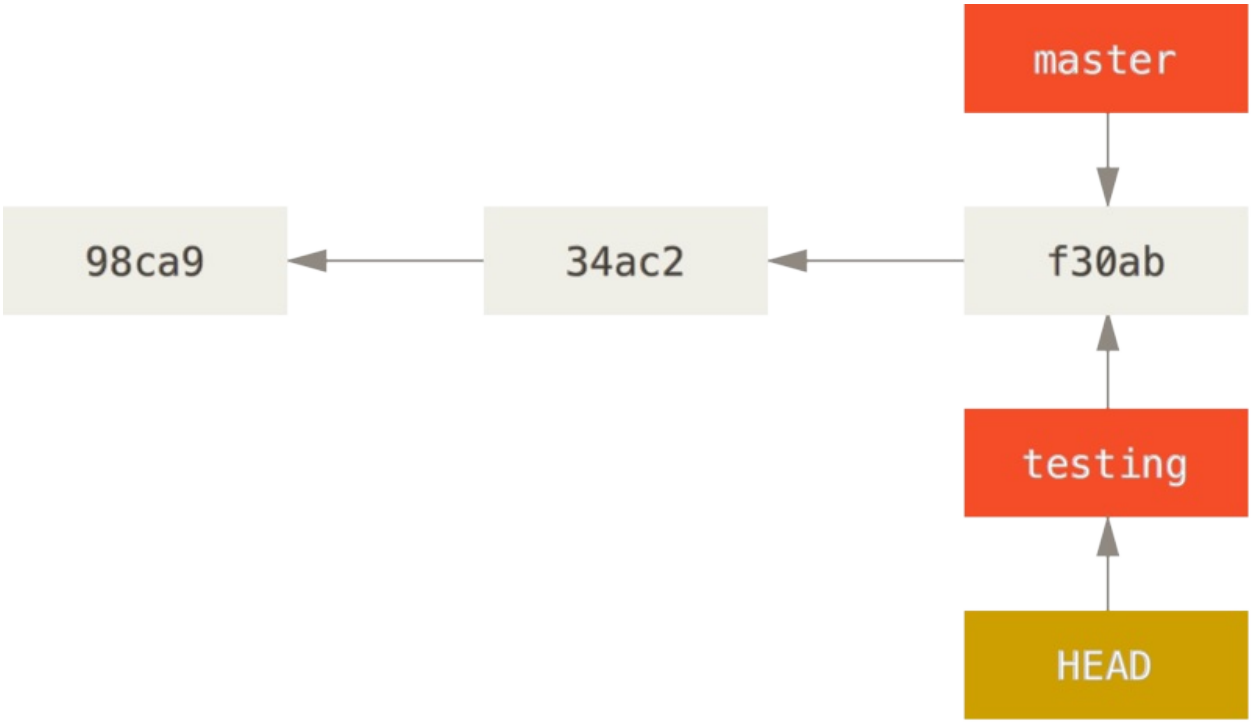


Figure 3-6. HEAD 指向当前所在的分支

那么，这样的实现方式会给我们带来什么好处呢？现在不妨再提交一次：

```
$ vim test.rb
$ git commit -a -m 'made a change'
```

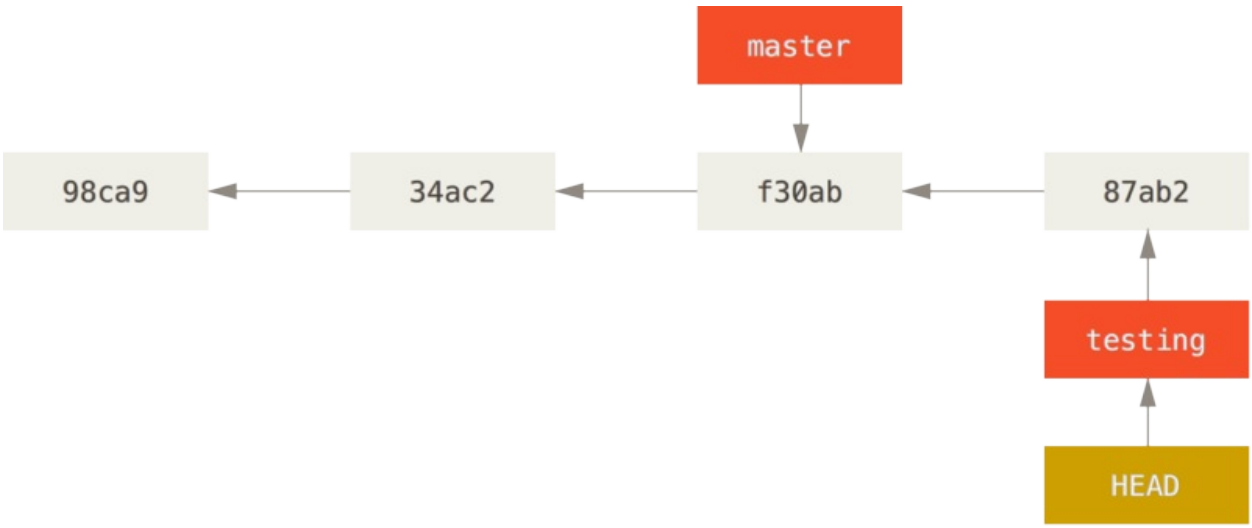


Figure 3-7. HEAD 分支随着提交操作自动向前移动

如图所示，你的 `testing` 分支向前移动了，但是 `master` 分支却没有，它仍然指向运行 `git checkout` 时所指的对象。这就有意思了，现在我们切换回 `master` 分支看看：

```
$ git checkout master
```

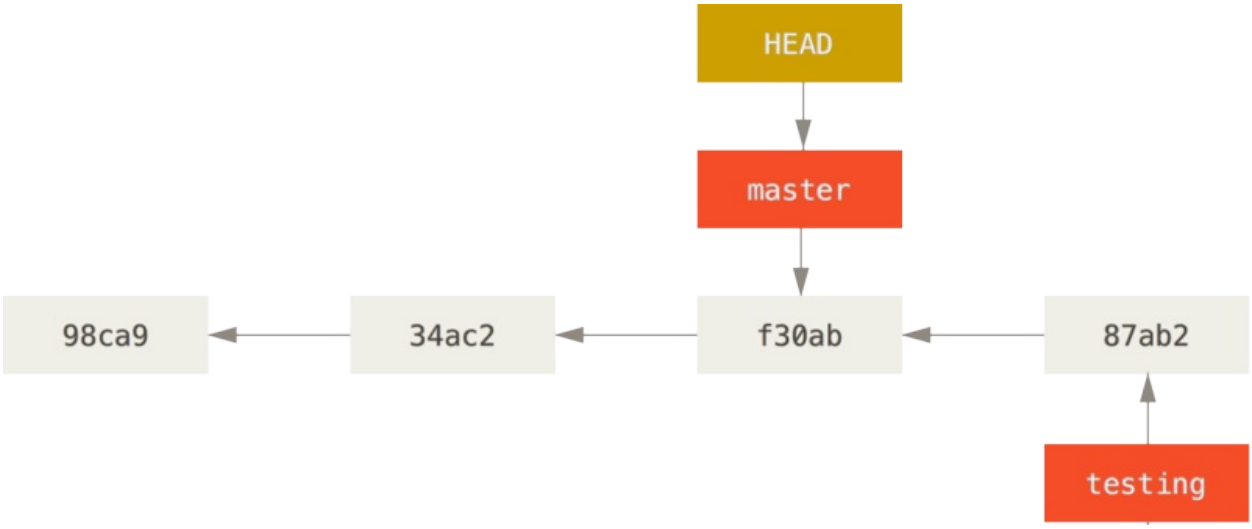


Figure 3-8. 检出时 HEAD 随之移动

这条命令做了两件事。一是使 HEAD 指回 `master` 分支，二是将工作目录恢复成 `master` 分支所指向的快照内容。也就是说，你现在做修改的话，项目将始于一个较旧的版本。本质上来讲，这就是忽略 `testing` 分支所做的修改，以便于向另一个方向进行开发。

> **分支切换会改变你工作目录中的文件**
>
> 在切换分支时，一定要注意你工作目录里的文件会被改变。如果是切换到一个较旧的分支，你的工作目录会恢复到该分支最后一次提交时的样子。如果 Git 不能干净利落地完成这个任务，它将禁止切换分支。

我们不妨再稍微做些修改并提交：

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

现在，这个项目的提交历史已经产生了分叉（参见 Figure 3-9）。因为刚才你创建了一个新分支，并切换过去进行了一些工作，随后又切换回 master 分支进行了另外一些工作。上述两次改动针对的是不同分支：你可以在不同分支间不断地来回切换和工作，并在时机成熟时将它们合并起来。而所有这些工作，你需要的命令只有 `branch`、`checkout` 和 `commit`。

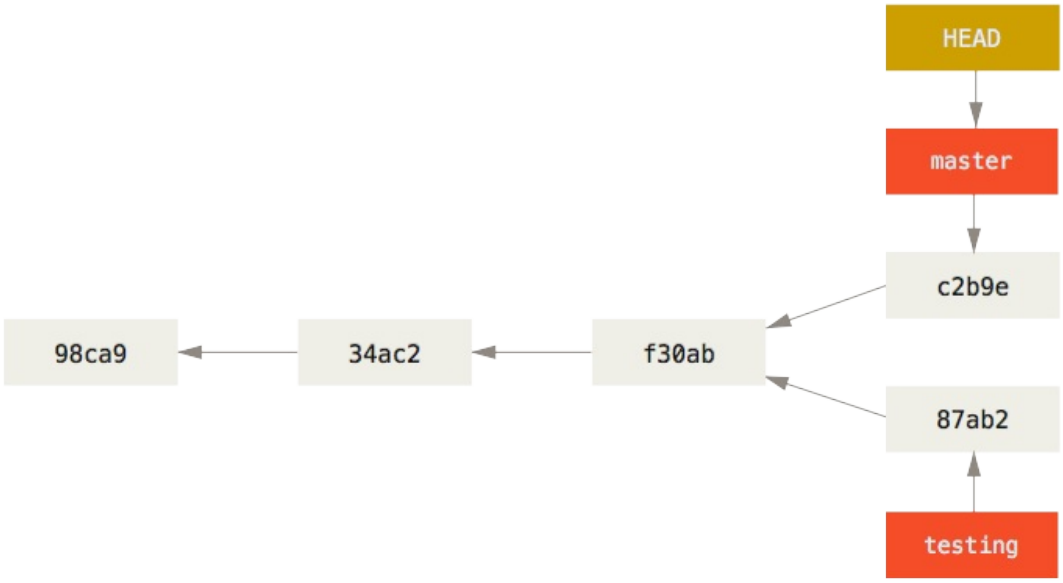


Figure 3-9. 项目分叉历史

你可以简单地使用 `git log` 命令查看分叉历史。运行 `git log --oneline --decorate --graph --all` ，它会输出你的提交历史、各个分支的指向以及项目的分支分叉情况。

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
| * 87ab2 (testing) made a change
|/
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

由于 Git 的分支实质上仅是包含所指对象校验和（长度为 40 的 SHA-1 值字符串）的文件，所以它的创建和销毁都异常高效。创建一个新分支就像是往一个文件中写入 41 个字节（40 个字符和 1 个换行符），如此的简单能不快吗？

这与过去大多数版本控制系统形成了鲜明的对比，它们在创建分支时，将所有的项目文件都复制一遍，并保存到一个特定的目录。完成这样繁琐的过程通常需要好几秒钟，有时甚至需要好几分钟。所需时间的长短，完全取决于项目的规模。而在 Git 中，任何规模的项目都能在瞬间创建新分支。同时，由于每次提交都会记录父对象，所以寻找恰当的合并基础（译注：即共同祖先）也是同样的简单和高效。这些高效的特性使得 Git 鼓励开发人员频繁地创建和使用分支。

接下来，让我们看看为什么你应该这么做？

3.2 分支的新建与合并

分支的新建与合并

让我们来看一个简单的分支新建与分支合并的例子，实际工作中你可能会用到类似的工作流。你将经历如下步骤：

1. 开发某个网站。
2. 为实现某个新的需求，创建一个分支。
3. 在这个分支上开展工作。

正在此时，你突然接到一个电话说有个很严重的问题需要紧急修补。你将按照如下方式来处理：

1. 切换到你的线上分支（production branch）。
2. 为这个紧急任务新建一个分支，并在其中修复它。
3. 在测试通过之后，切换回线上分支，然后合并这个修补分支，最后将改动推送到线上分支。
4. 切换回你最初工作的分支上，继续工作。

新建分支

首先，我们假设你正在你的项目上工作，并且已经有一些提交。

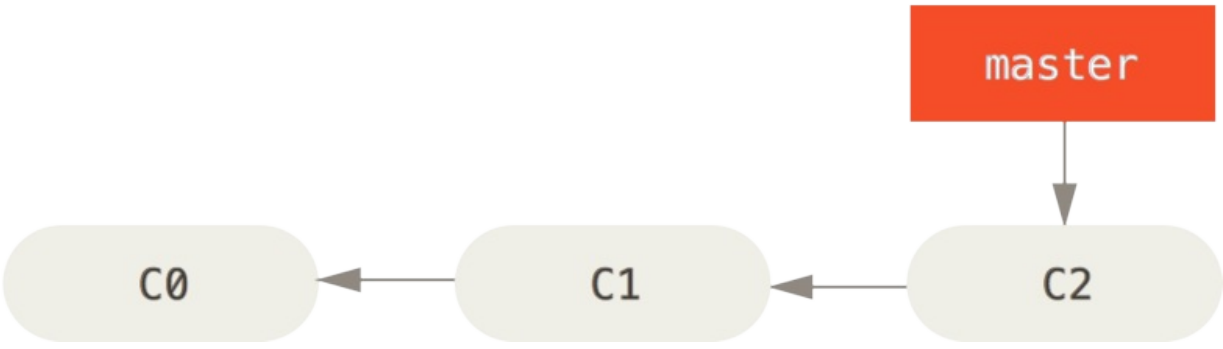


Figure 3-10. 一个简单提交历史

现在，你已经决定要解决你的公司使用的问题追踪系统中的 #53 问题。想要新建一个分支并同时切换到那个分支上，你可以运行一个带有 `-b` 参数的 `git checkout` 命令：

```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

它是下面两条命令的简写：

```
$ git branch iss53
$ git checkout iss53
```

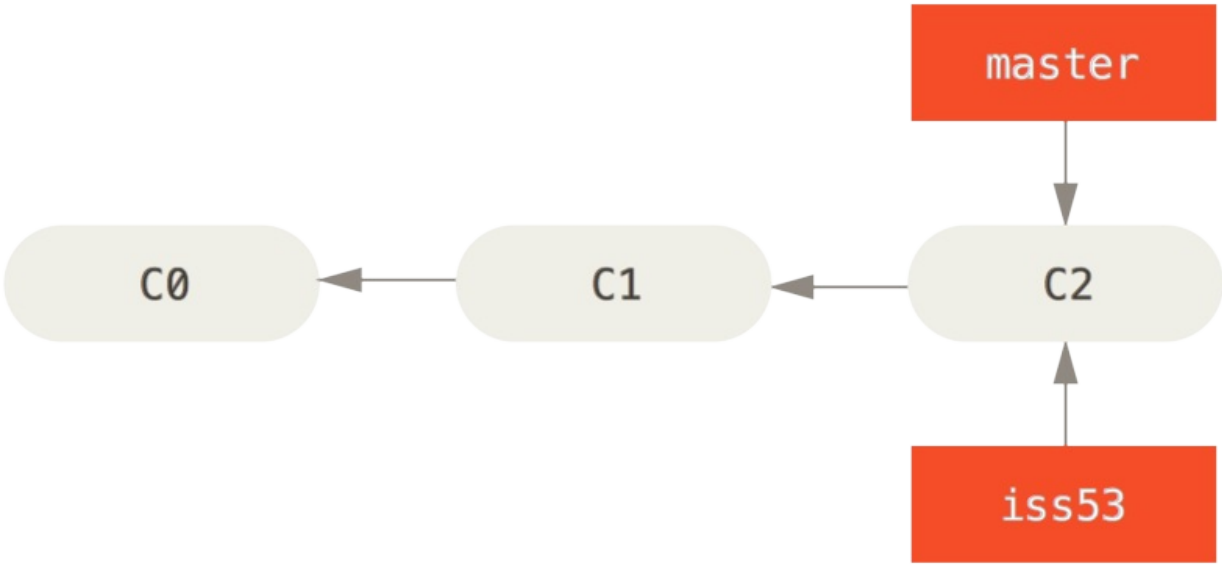


Figure 3-11. 创建一个新分支指针

你继续在 #53 问题上工作，并且做了一些提交。在此过程中，`iss53` 分支在不断的向前推进，因为你已经检出到该分支（也就是说，你的 `HEAD` 指针指向了 `iss53` 分支）

```
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
```

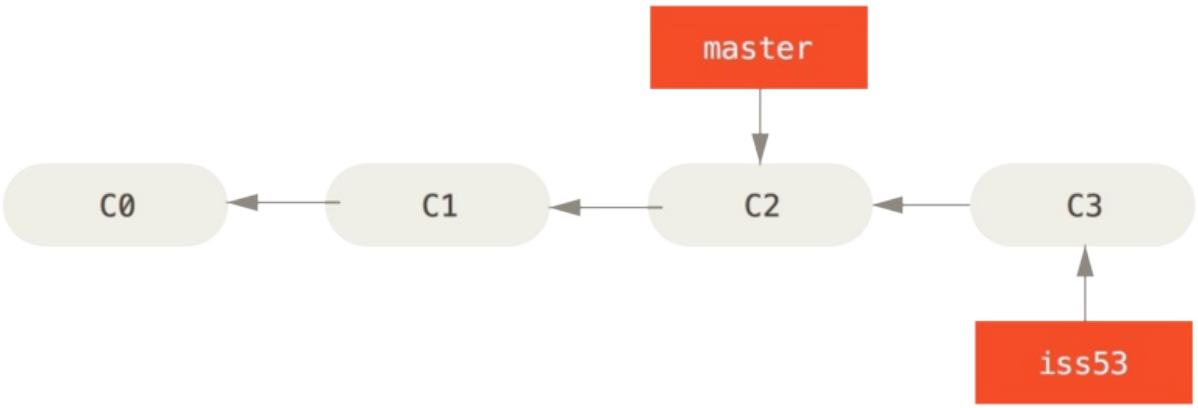


Figure 3-12. iss53 分支随着工作的进展向前推进

现在你接到那个电话，有个紧急问题等待你来解决。有了 Git 的帮助，你不必把这个紧急问题和 `iss53` 的修改混在一起，你也不需要花大力气来还原关于 53# 问题的修改，然后再添加关于这个紧急问题的修改，最后将这个修改提交到线上分支。你所要做的仅仅是切换回 `master` 分支。

但是，在你这么做之前，要留意你的工作目录和暂存区里那些还没有被提交的修改，它可能会和你即将检出的分支产生冲突从而阻止 Git 切换到该分支。最好的方法是，在你切换分支之前，保持好一个干净的状态。有一些方法可以绕过这个问题（即，保存进度（stashing）和 修补提交（commit amending）），我们会在 “储藏与清理” 中看到关于这两个命令的介绍。现在，我们假设你已经把你的修改全部提交了，这时你可以切换回 `master` 分支了：

```
$ git checkout master
Switched to branch 'master'
```

这个时候，你的工作目录和你在开始 #53 问题之前一模一样，现在你可以专心修复紧急问题了。请牢记：当你切换分支的时候，Git 会重置你的工作目录，使其看起来像回到了你在那个分支上最后一次提交的样子。Git 会自动添加、删除、修改文件以确保此时你的工作目录和这个分支最后一次提交时的样子一模一样。

接下来，你要修复这个紧急问题。让我们建立一个针对该紧急问题的分支（hotfix branch），在该分支上工作直到问题解决：

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 1fb7853] fixed the broken email address
 1 file changed, 2 insertions(+)
```

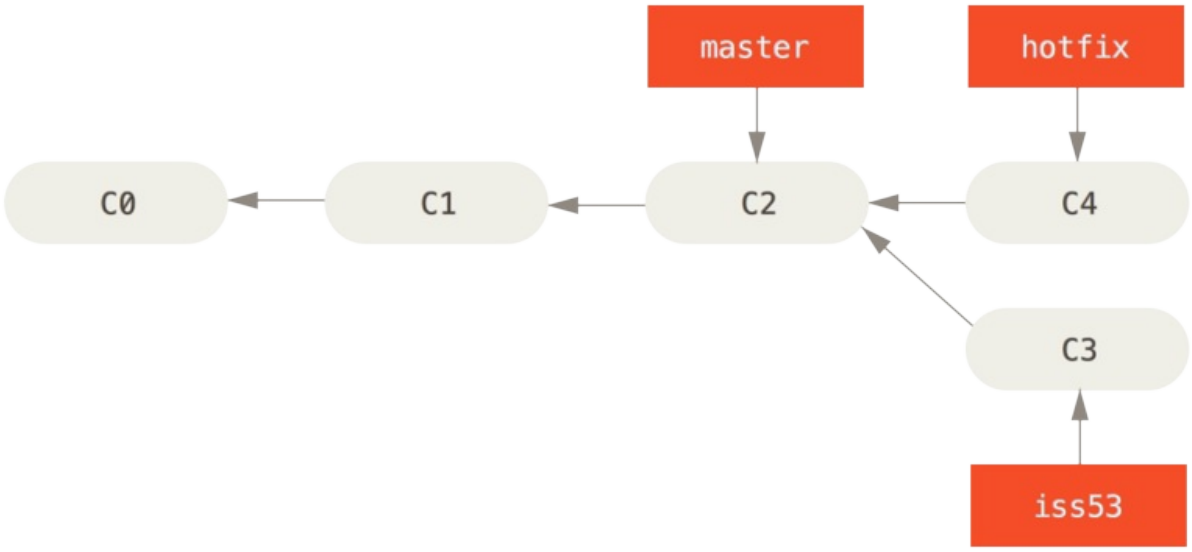


Figure 3-13. 基于 `master` 分支的紧急问题分支 `hotfix branch`

你可以运行你的测试，确保你的修改是正确的，然后将其合并回你的 `master` 分支来部署到线上。你可以使用 `git merge` 命令来达到上述目的：

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

在合并的时候，你应该注意到了"快进（fast-forward）"这个词。由于当前 `master` 分支所指向的提交是你当前提交（有关 hotfix 的提交）的直接上游，所以 Git 只是简单的将指针向前移动。换句话说，当你试图合并两个分支时，如果顺着一个分支走下去能够到达另一个分支，那么 Git 在合并两者的时候，只会简单的将指针向前推进（指针右移），因为这种情况下的合并操作没有需要解决的分歧——这就叫做 “快进（fast-forward）” 。

现在，最新的修改已经在 `master` 分支所指向的提交快照中，你可以着手发布该修复了。

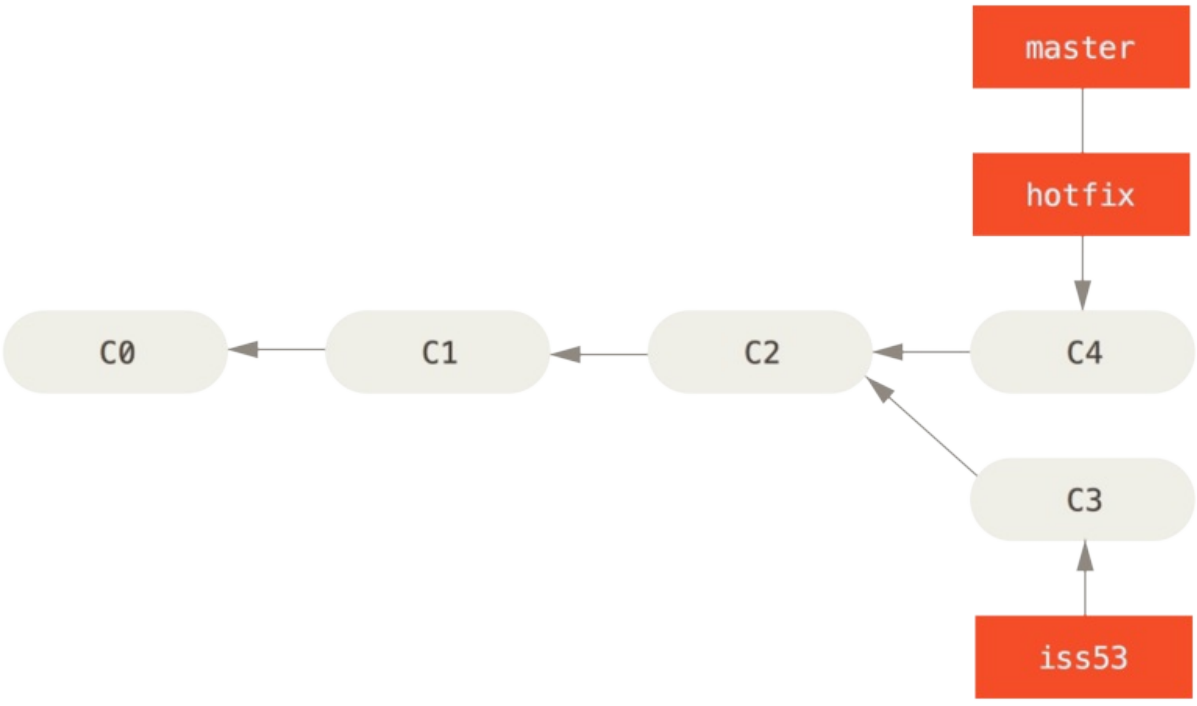


Figure 3-14. `master` 被快进到 `hotfix`

关于这个紧急问题的解决方案发布之后，你准备回到被打断之前时的工作中。然而，你应该先删除 `hotfix` 分支，因为你已经不再需要它了 —— `master` 分支已经指向了同一个位置。你可以使用带

`-d` 选项的 `git branch` 命令来删除分支：

```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

现在你可以切换回你正在工作的分支继续你的工作，也就是针对 #53 问题的那个分支（iss53 分支）。

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
1 file changed, 1 insertion(+)
```

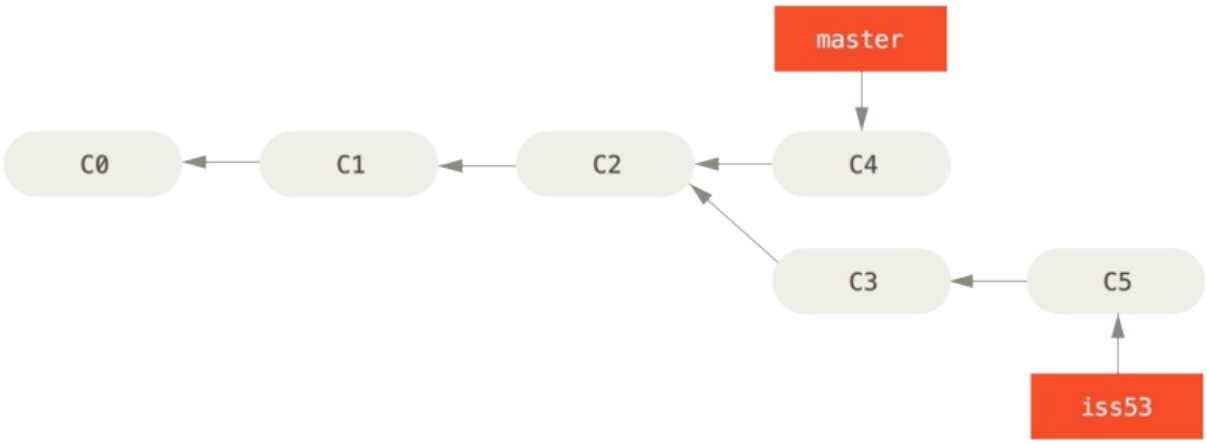


Figure 3-15. 继续在 `iss53` 分支上的工作

你在 `hotfix` 分支上所做的工作并没有包含到 `iss53` 分支中。如果你需要拉取 `hotfix` 所做的修改，你可以使用 `git merge master` 命令将 `master` 分支合并入 `iss53` 分支，或者你也可以等到 `iss53` 分支完成其使命，再将其合并回 `master` 分支。

分支的合并

假设你已经修正了 #53 问题，并且打算将你的工作合并入 `master` 分支。为此，你需要合并 `iss53` 分支到 `master` 分支，这和之前你合并 `hotfix` 分支所做的工作差不多。你只需要检出到你想合并入的分支，然后运行 `git merge` 命令：

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
```

```
 index.html |    1 +
 1 file changed, 1 insertion(+)
```

这和你之前合并 `hotfix` 分支的时候看起来有一点不一样。在这种情况下，你的开发历史从一个更早的地方开始分叉开来（diverged）。因为，`master` 分支所在提交并不是 `iss53` 分支所在提交的直接祖先，Git 不得不做一些额外的工作。出现这种情况的时候，Git 会使用两个分支的末端所指的快照（`C4` 和 `C5`）以及这两个分支的工作祖先（`C2`），做一个简单的三方合并。

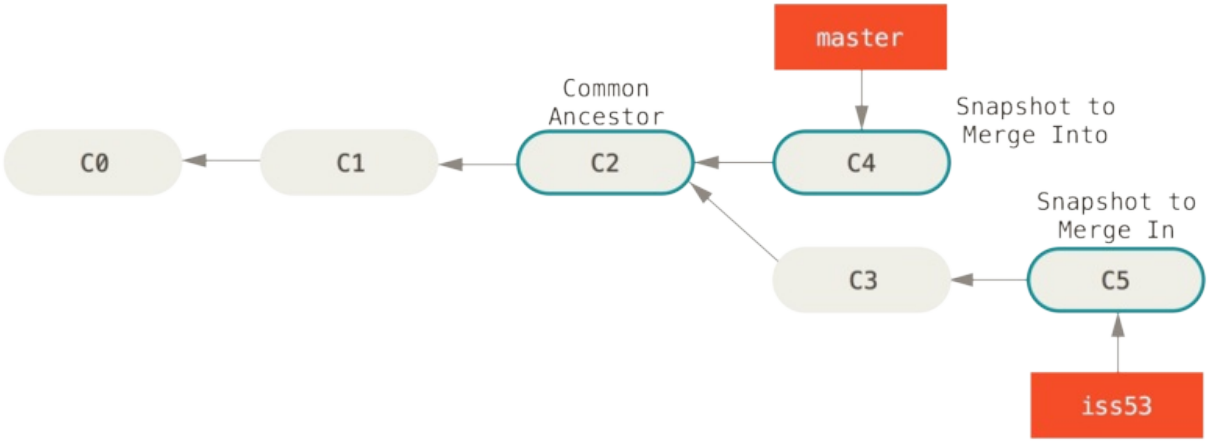


Figure 3-16. 一次典型合并中所用到的三个快照

和之间将分支指针向前推进所不同的是，Git 将此次三方合并的结果做了一个新的快照并且自动创建一个新的提交指向它。这个被称作一次合并提交，它的特别之处在于他有不止一个父提交。

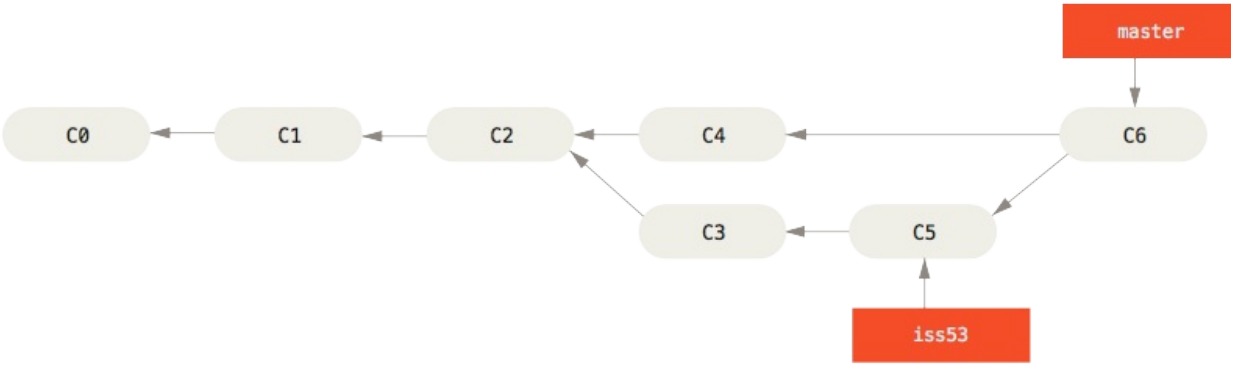


Figure 3-17. 一个合并提交

需要指出的是，Git 会自行决定选取哪一个提交作为最优的共同祖先，并以此作为合并的基础；这和更加古老的 CVS 系统或者 Subversion （1.5 版本之前）不同，在这些古老的版本管理系统中，用户需要自己选择最佳的合并基础。Git 的这个优势使其在合并操作上比其他系统要简单很多。

既然你的修改已经合并进来了，你已经不再需要 `iss53` 分支了。现在你可以在任务追踪系统中关闭此项任务，并删除这个分支。

```
$ git branch -d iss53
```

遇到冲突时的分支合并

有时候合并操作不会如此顺利。如果你在两个不同的分支中，对同一个文件的同一个部分进行了不同的修改，Git 就没法干净的合并它们。如果你对 #53 问题的修改和有关 `hotfix` 的修改都涉及到同一个文件的同一处，在合并它们的时候就会产生合并冲突：

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

此时 Git 做了合并，但是没有自动地创建一个新的合并提交。Git 会暂停下来，等待你去解决合并产生的冲突。你可以在合并冲突后的任意时刻使用 `git status` 命令来查看那些因包含合并冲突而处于未合并（unmerged）状态的文件：

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

任何因包含合并冲突而有待解决的文件，都会以未合并状态标识出来。Git 会在有冲突的文件中加入标准的冲突解决标记，这样你可以打开这些包含冲突的文件然后手动解决冲突。出现冲突的文件会包含一些特殊区段，看起来像下面这个样子：

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

这表示 `HEAD` 所指示的版本（也就是你的 `master` 分支所在的位置，因为你在运行 merge 命令的时候已经检出到了这个分支）在这个区段的上半部分（ `=======` 的上半部分），而 `iss53` 分支所指示的版本在 `=======` 的下半部分。为了解决冲突，你必须选择使用由 `=======` 分割的两部分中的一个，或者你也可以自行合并这些内容。例如，你可以通过把这段内容换成下面的样子来解决冲突：

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

上述的冲突解决方案仅保留了其中一个分支的修改，并且 `<<<<<<<` ， `=======` ，和 `>>>>>>>` 这些行被完全删除了。在你解决了所有文件里的冲突之后，对每个文件使用 `git add` 命令来将其标记为冲突已解决。一旦暂存这些原本有冲突的文件，Git 就会将它们标记为冲突已解决。

如果你想使用图形化工具来解决冲突，你可以运行 `git mergetool` ，该命令会为你启动一个合适的可视化合并工具，并带领你一步一步解决这些冲突：

```
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis bc3
codecompare vimdiff emerge
Merging:
index.html

Normal merge conflict for 'index.html':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

如果你想使用除默认工具（在这里 Git 使用 `opendiff` 做为默认的合并工具，因为作者在 Mac 上运行该程序）外的其他合并工具，你可以在 “下列工具中（one of the following tools）” 这句后面看到所有支持的合并工具。然后输入你喜欢的工具名字就可以了。

如果你需要更加高级的工具来解决复杂的合并冲突，我们会在 “高级合并” 介绍更多关于分支合并的内容。

等你退出合并工具之后，Git 会询问刚才的合并是否成功。如果你回答是，Git 会暂存那些文件以表明冲突已解决：你可以再次运行 `git status` 来确认所有的合并冲突都已被解决：

```
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)
```

```
Changes to be committed:

    modified:   index.html
```

如果你对结果感到满意，并且确定之前有冲突的的文件都已经暂存了，这时你可以输入 `git commit` 来完成合并提交。默认情况下提交信息看起来像下面这个样子：

```
Merge branch 'iss53'

Conflicts:
    index.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#       .git/MERGE_HEAD
# and try again.


# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#       modified:   index.html
#
```

如果你觉得上述的信息不够充分，不能完全体现分支合并的过程，你可以修改上述信息，添加一些细节给未来检视这个合并的读者一些帮助，告诉他们你是如何解决合并冲突的，以及理由是什么。

3.3 分支管理

分支管理

现在已经创建、合并、删除了一些分支，让我们看看一些常用的分支管理工具。

`git branch` 命令不只是可以创建与删除分支。如果不加任何参数运行它，会得到当前所有分支的一个列表：

```
$ git branch
  iss53
* master
  testing
```

注意 `master` 分支前的 `*` 字符：它代表现在检出的那一个分支（也就是说，当前 `HEAD` 指针所指向的分支）。这意味着如果在这时候提交，`master` 分支将会随着新的工作向前移动。如果需要查看每一个分支的最后一次提交，可以运行 `git branch -v` 命令：

```
$ git branch -v
  iss53   93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

`--merged` 与 `--no-merged` 这两个有用的选项可以过滤这个列表中已经合并或尚未合并到当前分支的分支。如果要查看哪些分支已经合并到当前分支，可以运行 `git branch --merged`：

```
$ git branch --merged
  iss53
* master
```

因为之前已经合并了 `iss53` 分支，所以现在看到它在列表中。在这个列表中分支名字前没有 `*` 号的分支通常可以使用 `git branch -d` 删除掉；你已经将它们的工作整合到了另一个分支，所以并不会失去任何东西。

查看所有包含未合并工作的分支，可以运行 `git branch --no-merged`：

```
$ git branch --no-merged
  testing
```

这里显示了其他分支。因为它包含了还未合并的工作，尝试使用 `git branch -d` 命令删除它时会失败：

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

如果真的想要删除分支并丢掉那些工作，如同帮助信息里所指出的，可以使用 `-D` 选项强制删除它。

3.4 分支开发工作流

分支开发工作流

现在你已经学会新建和合并分支，那么你可以或者应该用它来做些什么呢？在本节，我们会介绍一些常见的利用分支进行开发的工作流程。而正是由于分支管理的便捷，才衍生出这些典型的工作模式，你可以根据项目实际情况选择一种用用看。

长期分支

因为 Git 使用简单的三方合并，所以就算在一段较长的时间内，反复把一个分支合并入另一个分支，也不是什么难事。也就是说，在整个项目开发周期的不同阶段，你可以同时拥有多个开放的分支；你可以定期地把某些特性分支合并入其他分支中。

许多使用 Git 的开发者都喜欢使用这种方式来工作，比如只在 `master` 分支上保留完全稳定的代码——有可能仅仅是已经发布或即将发布的代码。他们还有一些名为 `develop` 或者 `next` 的平行分支，被用来做后续开发或者测试稳定性——这些分支不必保持绝对稳定，但是一旦达到稳定状态，它们就可以被合并入 `master` 分支了。这样，在确保这些已完成的特性分支（短期分支，比如之前的 `iss53` 分支）能够通过所有测试，并且不会引入更多 bug 之后，就可以合并入主干分支中，等待下一次的发布。

事实上我们刚才讨论的，是随着你的提交而不断右移的指针。稳定分支的指针总是在提交历史中落后一大截，而前沿分支的指针往往比较靠前。

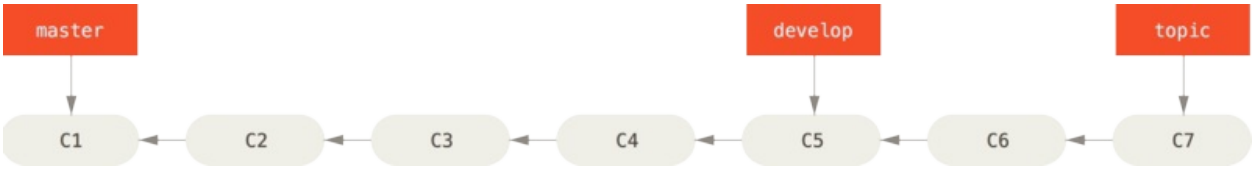


Figure 3-18. 渐进稳定分支的线性图

通常把他们想象成流水线（work silos）可能更好理解一点，那些经过测试考验的提交会被遴选到更加稳定的流水线上去。

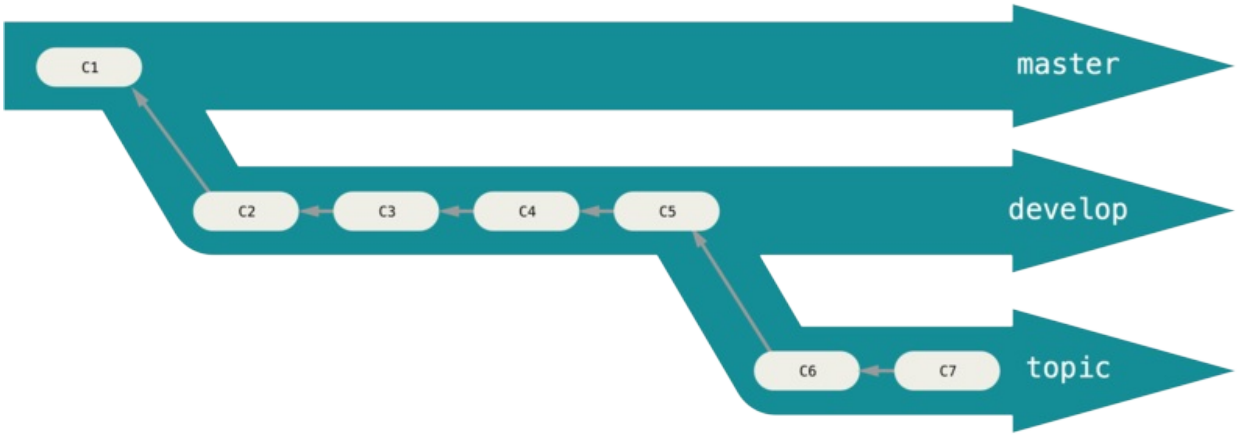


Figure 3-19. 渐进稳定分支的流水线（ “silo” ）视图

你可以用这种方法维护不同层次的稳定性。一些大型项目还有一个 `proposed`（建议）或 `pu: proposed updates`（建议更新）分支，它可能因包含一些不成熟的内容而不能进入 `next` 或者 `master` 分支。这么做的目的是使你的分支具有不同级别的稳定性；当它们具有一定程度的稳定性后，再把它们合并入具有更高级别稳定性的分支中。再次强调一下，使用多个长期分支的方法并非必要，但是这么做通常很有帮助，尤其是当你在一个非常庞大或者复杂的项目中工作时。

特性分支

特性分支对任何规模的项目都适用。特性分支是一种短期分支，它被用来实现单一特性或其相关工作。也许你从来没有在其他的版本控制系统（ `VCS` ）上这么做过，因为在那些版本控制系统中创建和合并分支通常很费劲。然而，在 Git 中一天之内多次创建、使用、合并、删除分支都很常见。

你已经在上一节中你创建的 `iss53` 和 `hotfix` 特性分支中看到过这种用法。你在上一节用到的特性分支（ `iss53` 和 `hotfix` 分支）中提交了一些更新，并且在它们合并入主干分支之后，你又删除了它们。这项技术能使你快速并且完整地进行上下文切换（context-switch）——因为你的工作被分散到不同的流水线中，在不同的流水线中每个分支都仅与其目标特性相关，因此，在做代码审查之类的工作的时候就能更加容易地看出你做了哪些改动。你可以把做出的改动在特性分支中保留几分钟、几天甚至几个月，等它们成熟之后再合并，而不用在乎它们建立的顺序或工作进度。

考虑这样一个例子，你在 `master` 分支上工作到 `C1`，这时为了解决一个问题而新建 `iss91` 分支，在 `iss91` 分支上工作到 `C4`，然而对于那个问题你又有了新的想法，于是你再新建一个 `iss91v2` 分支试图用另一种方法解决那个问题，接着你回到 `master` 分支工作了一会儿，你又冒出了一个不太确定的想法，你便在 `C10` 的时候新建一个 `dumbidea` 分支，并在上面做些实验。你的提交历史看起来像下面这个样子：

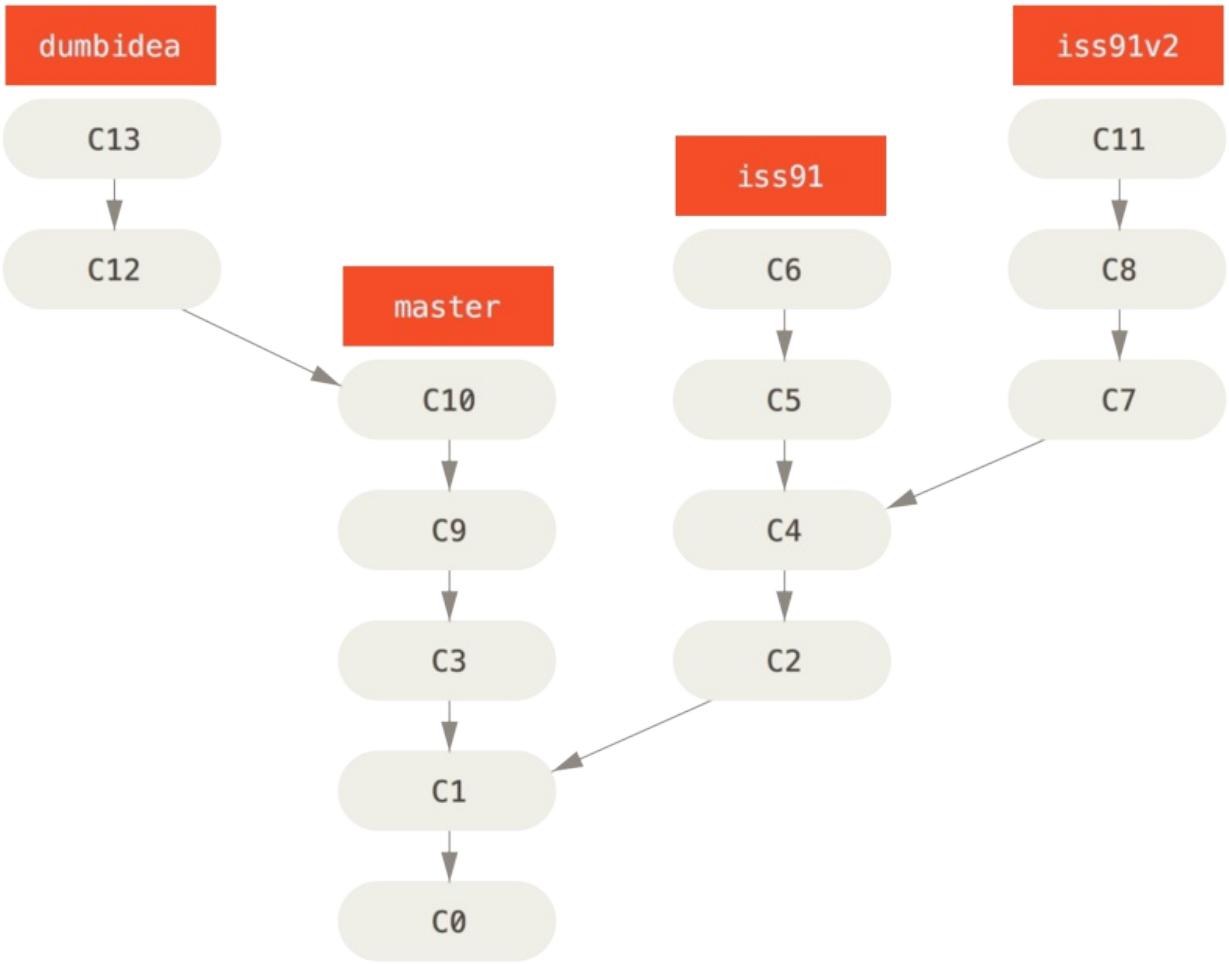


Figure 3-20. 拥有多个特性分支的提交历史

现在，我们假设两件事情：你决定使用第二个方案来解决那个问题，即使用在 `iss91v2` 分支中方案；另外，你将 `dumbidea` 分支拿给你的同事看过之后，结果发现这是个惊人之举。这时你可以抛弃 `iss91` 分支（即丢弃 `C5` 和 `C6` 提交），然后把另外两个分支合并入主干分支。最终你的提交历史看起来像下面这个样子：

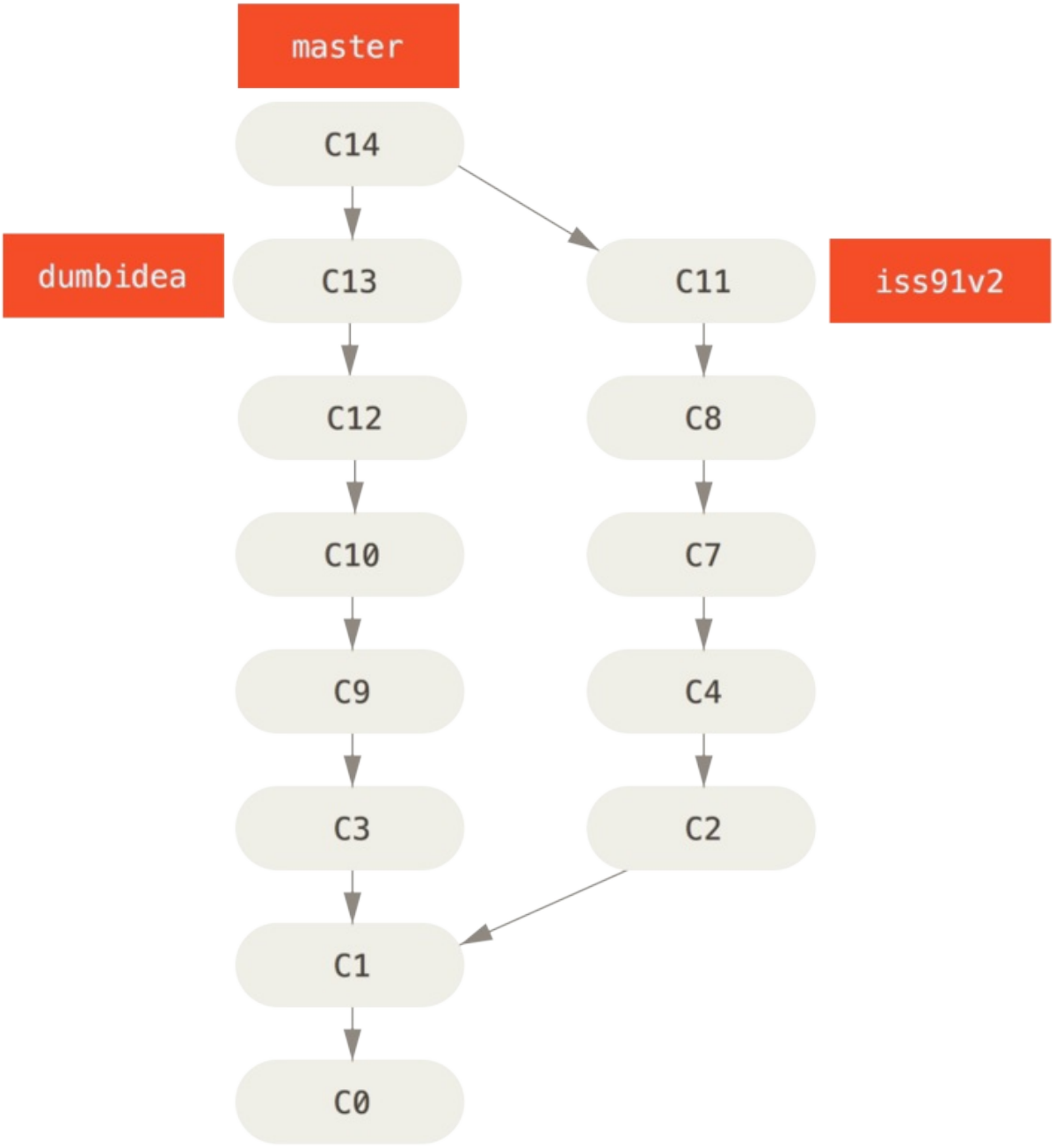


Figure 3-21. 合并了 `dumbidea` 和 `iss91v2` 分支之后的提交历史

我们将会在 Chapter 5 中向你揭示更多有关分支工作流的细节，因此，请确保你阅读完那个章节之后，再来决定你的下个项目要使用什么样的分支策略（branching scheme）。

请牢记，当你做这么多操作的时候，这些分支全部都存于本地。当你新建和合并分支的时候，所有这一切都只发生在你本地的 Git 版本库中 —— 没有与服务器发生交互。

3.5 远程分支

远程分支

远程引用是对远程仓库的引用（指针），包括分支、标签等等。你可以通过 `git ls-remote (remote)` 来显式地获得远程引用的完整列表，或者通过 `git remote show (remote)` 获得远程分支的更多信息。然而，一个更常见的做法是利用远程跟踪分支。

远程跟踪分支是远程分支状态的引用。它们是你不能移动的本地引用，当你做任何网络通信操作时，它们会自动移动。远程跟踪分支像是你上次连接到远程仓库时，那些分支所处状态的书签。

它们以 `(remote)/(branch)` 形式命名。例如，如果你想要看你最后一次与远程仓库 `origin` 通信时 `master` 分支的状态，你可以查看 `origin/master` 分支。你与同事合作解决一个问题并且他们推送了一个 `iss53` 分支，你可能有自己的本地 `iss53` 分支；但是在服务器上的分支会指向 `origin/iss53` 的提交。

这可能有一点儿难以理解，让我们来看一个例子。假设你的网络里有一个在 `git.ourcompany.com` 的 Git 服务器。如果你从这里克隆，Git 的 `clone` 命令会为你自动将其命名为 `origin`，拉取它的所有数据，创建一个指向它的 `master` 分支的指针，并且在本地将其命名为 `origin/master`。Git 也会给你一个与 origin 的 `master` 分支在指向同一个地方的本地 `master` 分支，这样你就有工作的基础。

> “origin” 并无特殊含义

> 远程仓库名字 “origin” 与分支名字 “master” 一样，在 Git 中并没有任何特别的含义一样。同时 “master” 是当你运行 `git init` 时默认的起始分支名字，原因仅仅是它的广泛使用，“origin” 是当你运行 `git clone` 时默认的远程仓库名字。如果你运行 `git clone -o booyah`，那么你默认的远程分支名字将会是 `booyah/master`。

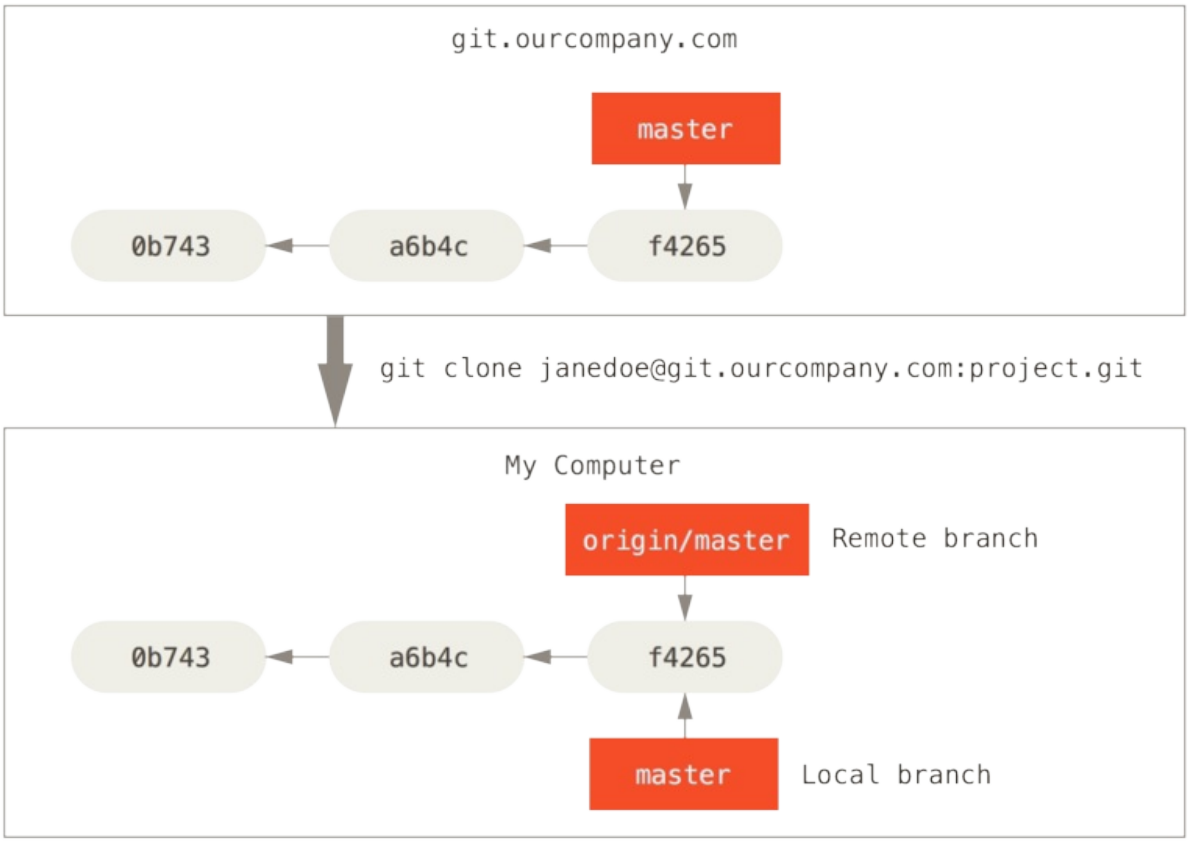


Figure 3-22. 克隆之后的服务器与本地仓库

如果你在本地的 `master` 分支做了一些工作，然而在同一时间，其他人推送提交到 `git.ourcompany.com` 并更新了它的 `master` 分支，那么你的提交历史将向不同的方向前进。也许，只要你不与 origin 服务器连接，你的 `origin/master` 指针就不会移动。

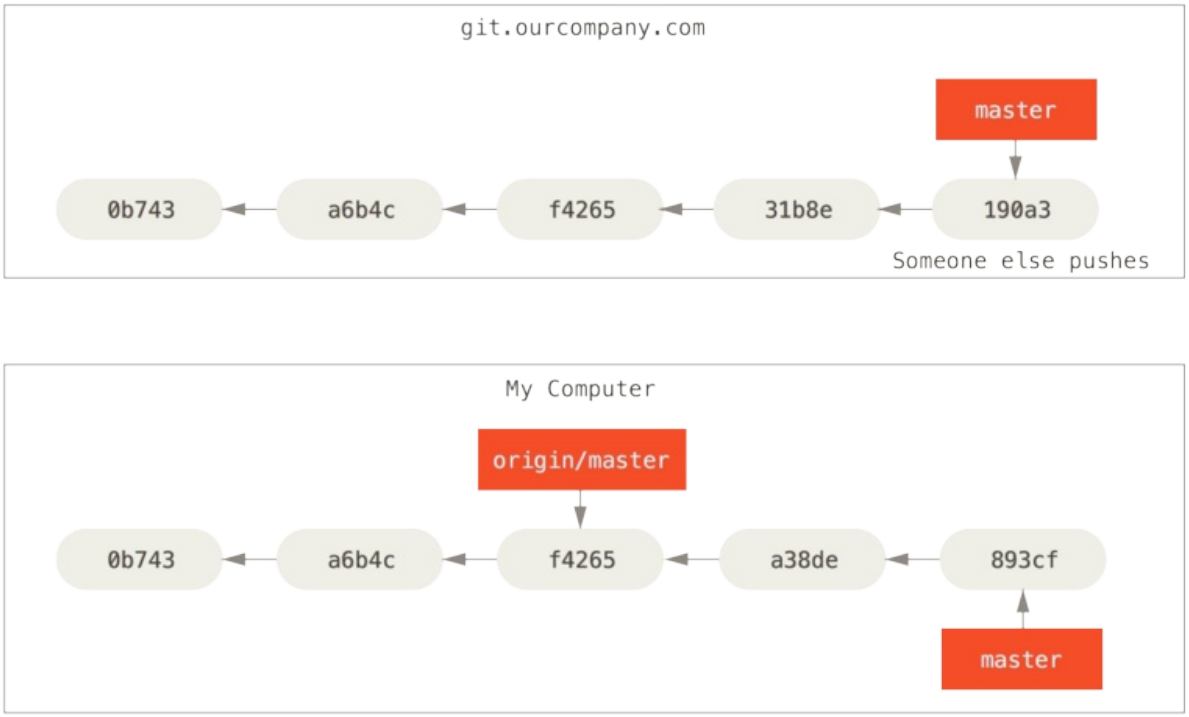


Figure 3-23. 本地与远程的工作可以分叉

如果要同步你的工作，运行 `git fetch origin` 命令。这个命令查找 “origin” 是哪一个服务器（在本例中，它是 `git.ourcompany.com`），从中抓取本地没有的数据，并且更新本地数据库，移动 `origin/master` 指针指向新的、更新后的位置。

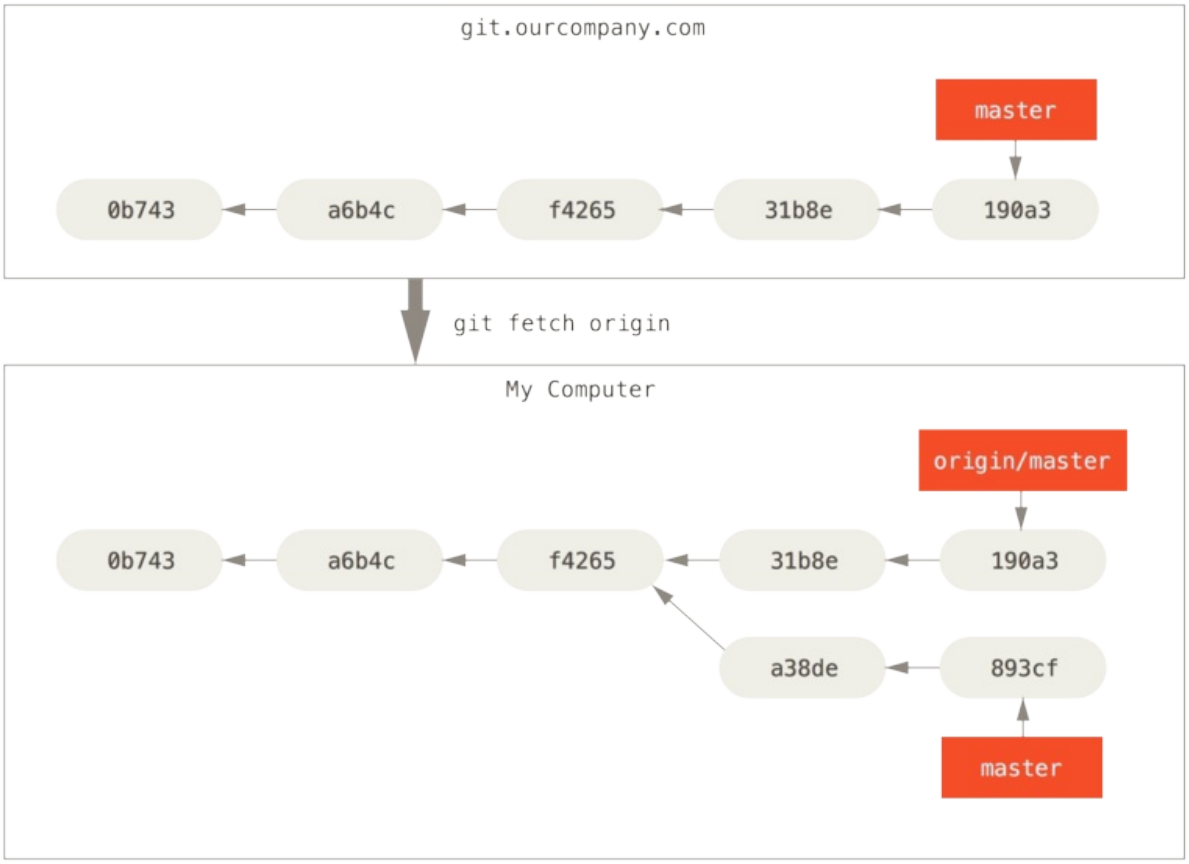


Figure 3-24. `git fetch` 更新你的远程仓库引用

为了演示有多个远程仓库与远程分支的情况，我们假定你有另一个内部 Git 服务器，仅用于你的 sprint 小组的开发工作。这个服务器位于 `git.team1.ourcompany.com`。你可以运行 `git remote add` 命令添加一个新的远程仓库引用到当前的项目，这个命令我们会在 Chapter 2 中详细说明。将这个远程仓库命名为 `teamone`，将其作为整个 URL 的缩写。

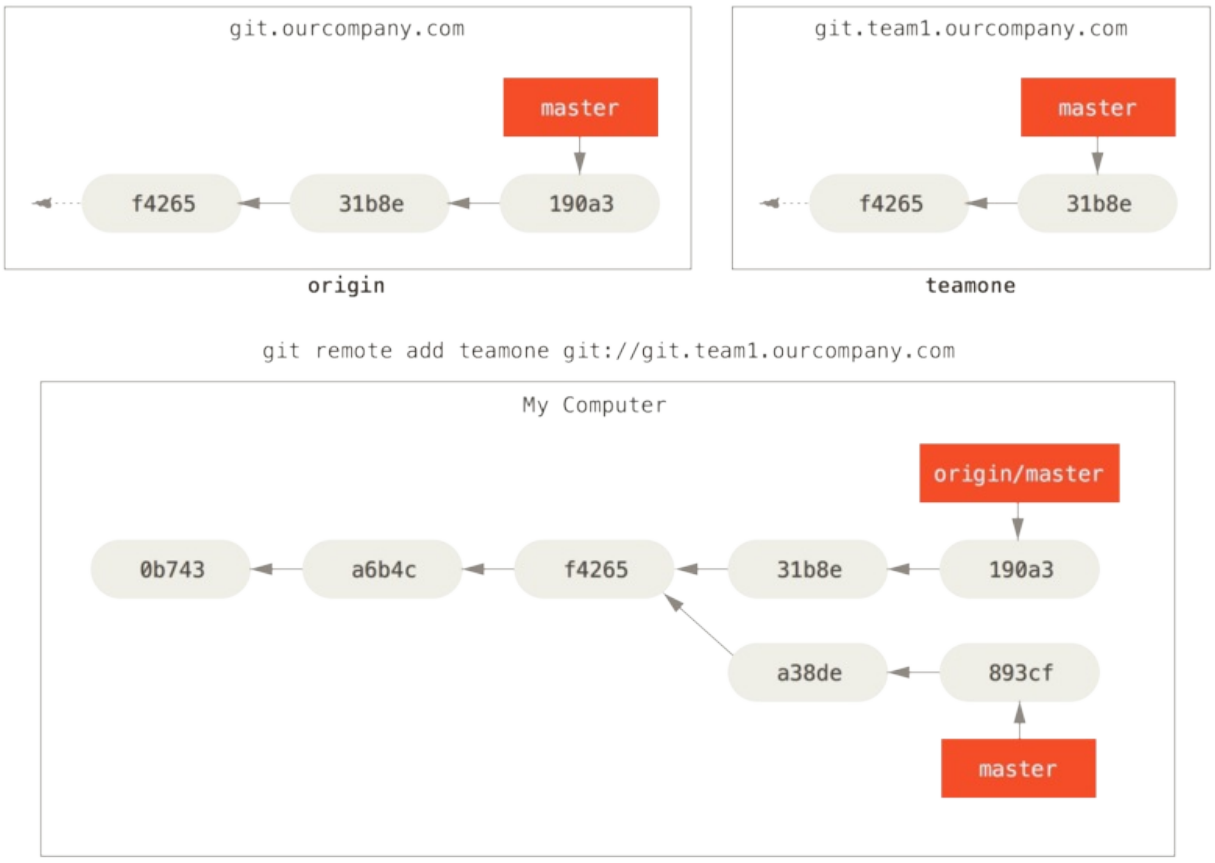


Figure 3-25. 添加另一个远程仓库

现在，可以运行 `git fetch teamone` 来抓取远程仓库 `teamone` 有而本地没有的数据。因为那台服务器上现有的数据是 `origin` 服务器上的一个子集，所以 Git 并不会抓取数据而是会设置远程跟踪分支 `teamone/master` 指向 `teamone` 的 `master` 分支。

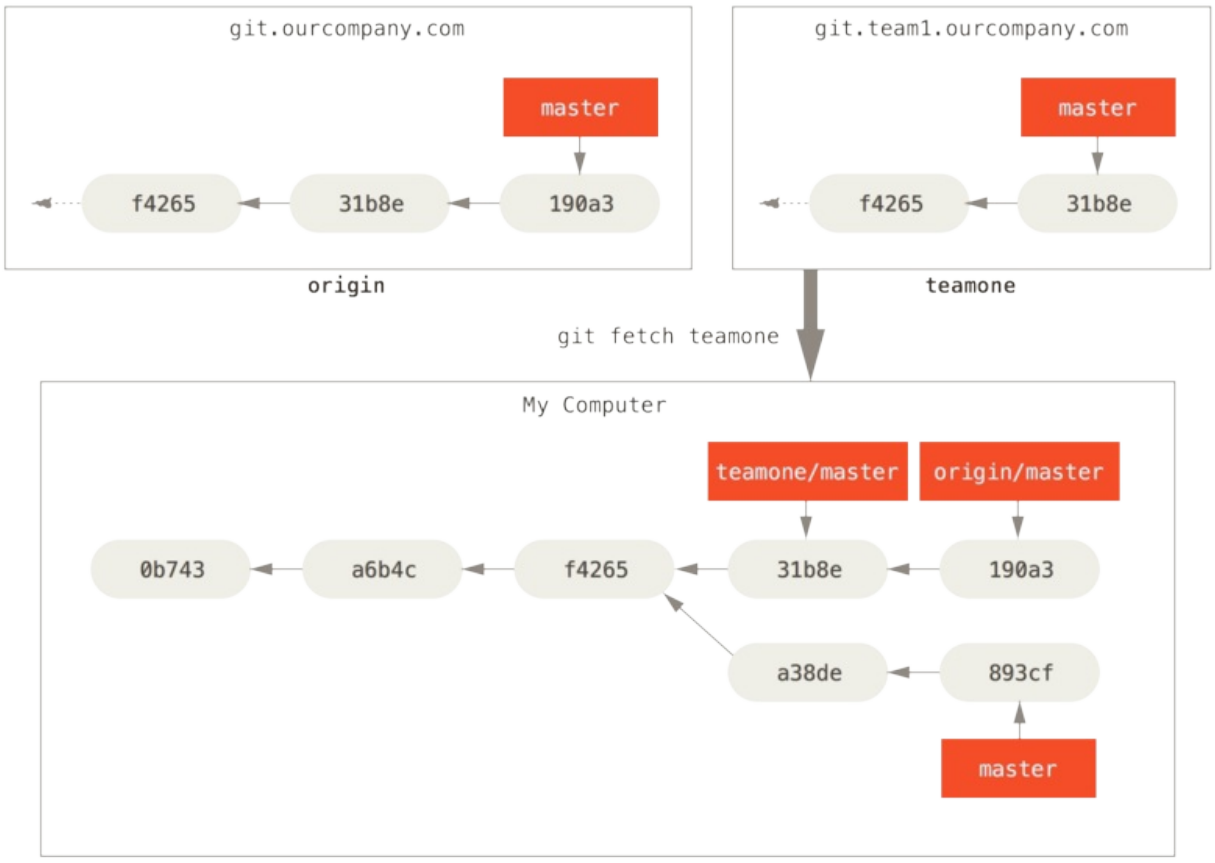


Figure 3-26. 远程跟踪分支 `teamone/master`

推送

当你想要公开分享一个分支时，需要将其推送到有写入权限的远程仓库上。本地的分支并不会自动与远程仓库同步 - 你必须显式地推送想要分享的分支。这样，你就可以把不愿意分享的内容放到私人分支上，而将需要和别人协作的内容推送到公开分支。

如果希望和别人一起在名为 `serverfix` 的分支上工作，你可以像推送第一个分支那样推送它。运行 `git push (remote) (branch)`：

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
 * [new branch]      serverfix -> serverfix
```

这里有些工作被简化了。Git 自动将 `serverfix` 分支名字展开为 `refs/heads/serverfix:refs/heads/serverfix`，那意味着， “推送本地的 serverfix 分支来更

新远程仓库上的 serverfix 分支。” 我们将会详细学习 Chapter 10 的 `refs/heads/` 部分，但是现在可以先把它放在儿。你也可以运行 `git push origin serverfix:serverfix`，它会做同样的事 - 相当于它说，“推送本地的 serverfix 分支，将其作为远程仓库的 serverfix 分支” 可以通过这种格式来推送本地分支到一个命名不相同的远程分支。如果并不想让远程仓库上的分支叫做 `serverfix`，可以运行 `git push origin serverfix:awesomebranch` 来将本地的 `serverfix` 分支推送到远程仓库上的 `awesomebranch` 分支。

> 如何避免每次输入密码
>
> 如果你正在使用 HTTPS URL 来推送，Git 服务器会询问用户名与密码。默认情况下它会在终端中提示服务器是否允许你进行推送。

如果不想在每一次推送时都输入用户名与密码，你可以设置一个 “credential cache” 。最简单的方式就是将其保存在内存中几分钟，可以简单地运行 `git config --global credential.helper cache` 来设置它。

想要了解更多关于不同验证缓存的可用选项，查看 “凭证存储” 。

下一次其他协作者从服务器上抓取数据时，他们会在本地生成一个远程分支 `origin/serverfix`，指向服务器的 `serverfix` 分支的引用：

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

要特别注意的一点是当抓取到新的远程跟踪分支时，本地不会自动生成一份可编辑的副本（拷贝）。换一句话说，这种情况下，不会有一个新的 `serverfix` 分支 - 只有一个不可以修改的 `origin/serverfix` 指针。

可以运行 `git merge origin/serverfix` 将这些工作合并到当前所在的分支。如果想要在自己的 `serverfix` 分支上工作，可以将其建立在远程跟踪分支之上：

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

这会给你一个用于工作的本地分支，并且起点位于 `origin/serverfix`。

跟踪分支

从一个远程跟踪分支检出一个本地分支会自动创建一个叫做 “跟踪分支”（有时候也叫做 “上游分支”）。跟踪分支是与远程分支有直接关系的本地分支。如果在一个跟踪分支上输入 `git pull`，Git 能自动地识别去哪个服务器上抓取、合并到哪个分支。

当克隆一个仓库时，它通常会自动地创建一个跟踪 `origin/master` 的 `master` 分支。然而，如果你愿意的话可以设置其他的跟踪分支 - 其他远程仓库上的跟踪分支，或者不跟踪 `master` 分支。最简单的就是之前看到的例子，运行 `git checkout -b [branch] [remotename]/[branch]`。这是一个十分常用的操作所以 Git 提供了 `--track` 快捷方式：

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

如果想要将本地分支与远程分支设置为不同名字，你可以轻松地增加一个不同名字的本地分支的上一个命令：

```
$ git checkout -b sf origin/serverfix
Branch sf set up to track remote branch serverfix from origin.
Switched to a new branch 'sf'
```

现在，本地分支 `sf` 会自动从 `origin/serverfix` 拉取。

设置已有的本地分支跟踪一个刚刚拉取下来的远程分支，或者想要修改正在跟踪的上游分支，你可以在任意时间使用 `-u` 或 `--set-upstream-to` 选项运行 `git branch` 来显式地设置。

```
$ git branch -u origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
```

> 上游快捷方式
>
> 当设置好跟踪分支后，可以通过 `@{upstream}` 或 `@{u}` 快捷方式来引用它。所以在 `master` 分支时并且它正在跟踪 `origin/master` 时，如果愿意的话可以使用 `git merge @{u}` 来取代 `git merge origin/master`。

如果想要查看设置的所有跟踪分支，可以使用 `git branch` 的 `-vv` 选项。这会将所有的本地分支列出来并且包含更多的信息，如每一个分支正在跟踪哪个远程分支与本地分支是否是领先、落后或是都有。

```
$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] forgot the brackets
  master    1ae2a45 [origin/master] deploying index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it
  testing   5ea463a trying something new
```

这里可以看到 `iss53` 分支正在跟踪 `origin/iss53` 并且 “ahead” 是 2，意味着本地有两个提交还没有推送到服务器上。也能看到 `master` 分支正在跟踪 `origin/master` 分支并且是最新的。接下来可以看到 `serverfix` 分支正在跟踪 `teamone` 服务器上的 `server-fix-good` 分支并且领先 2 落后 1，意味着服务器上有一次提交还没有合并入同时本地有三次提交还没有推送。最后看到 `testing` 分支并没有跟踪任何远程分支。

需要重点注意的一点是这些数字的值来自于你从每个服务器上最后一次抓取的数据。这个命令并没有连接服务器，它只会告诉你关于本地缓存的服务器数据。如果想要统计最新的领先与落后数字，需要在运行此命令前抓取所有的远程仓库。可以像这样做：`$ git fetch --all; git branch -vv`

拉取

当 `git fetch` 命令从服务器上抓取本地没有的数据时，它并不会修改工作目录中的内容。它只会获取数据然后让你自己合并。然而，有一个命令叫作 `git pull` 在大多数情况下它的含义是一个 `git fetch` 紧接着一个 `git merge` 命令。如果有一个像之前章节中演示的设置好的跟踪分支，不管它是显式地设置还是通过 `clone` 或 `checkout` 命令为你创建的，`git pull` 都会查找当前分支所跟踪的服务器与分支，从服务器上抓取数据然后尝试合并入那个远程分支。

由于 `git pull` 的魔法经常令人困惑所以通常单独显式地使用 `fetch` 与 `merge` 命令会更好一些。

删除远程分支

假设你已经通过远程分支做完所有的工作了 - 也就是说你和你的协作者已经完成了一个特性并且将其合并到了远程仓库的 `master` 分支（或任何其他稳定代码分支）。可以运行带有 `--delete` 选项的 `git push` 命令来删除一个远程分支。如果想要从服务器上删除 `serverfix` 分支，运行下面的命令：

```
$ git push origin --delete serverfix
To https://github.com/schacon/simplegit
 - [deleted]         serverfix
```

基本上这个命令做的只是从服务器上移除这个指针。Git 服务器通常会保留数据一段时间直到垃圾回收运行，所以如果不小心删除掉了，通常是很容易恢复的。

3.6 变基

变基

在 Git 中整合来自不同分支的修改主要有两种方法： `merge` 以及 `rebase` 。在本节中我们将学习什么是“变基”，怎样使用“变基”，并将展示该操作的惊艳之处，以及指出在何种情况下你应避免使用它。

变基的基本操作

请回顾之前在 “分支的合并” 中的一个例子，你会看到开发任务分叉到两个不同分支，又各自提交了更新。

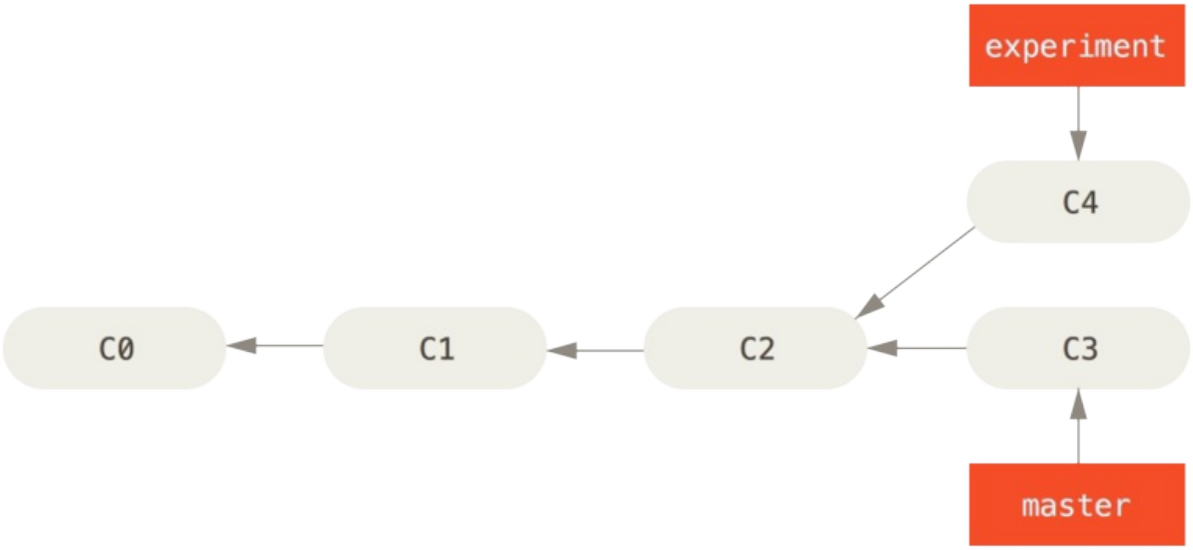


Figure 3-27. 分叉的提交历史

之前介绍过，整合分支最容易的方法是 `merge` 命令。它会把两个分支的最新快照（ `C3` 和 `C4` ）以及二者最近的共同祖先（ `C2` ）进行三方合并，合并的结果是生成一个新的快照（并提交）。

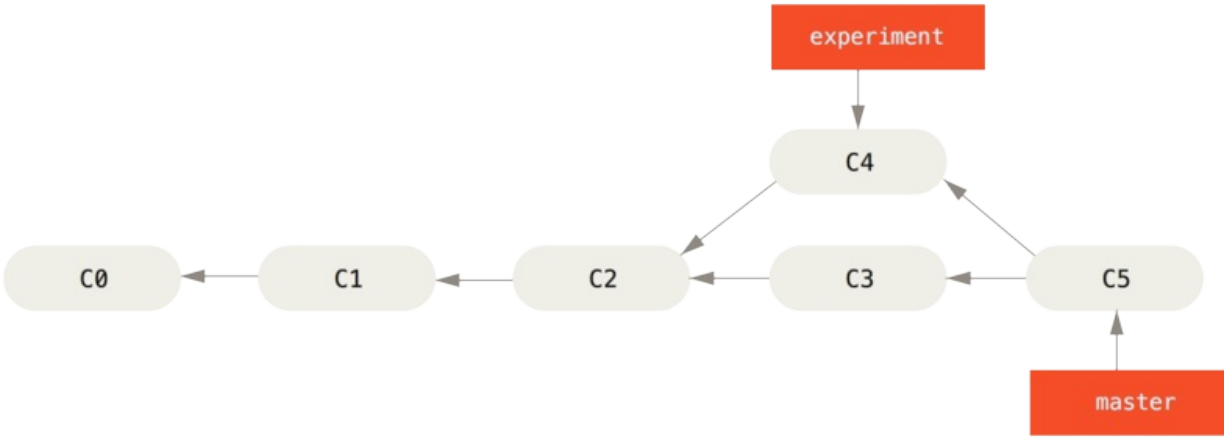


Figure 3-28. 通过合并操作来整合分叉了的历史

其实，还有一种方法：你可以提取在 `C4` 中引入的补丁和修改，然后在 `C3` 的基础上再应用一次。在 Git 中，这种操作就叫做 变基。你可以使用 `rebase` 命令将提交到某一分支上的所有修改都移至另一分支上，就好像“重新播放”一样。

在上面这个例子中，运行：

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

它的原理是首先找到这两个分支（即当前分支 `experiment`、变基操作的目标基底分支 `master`）的最近共同祖先 `C2`，然后对比当前分支相对于该祖先的历次提交，提取相应的修改并存为临时文件，然后将当前分支指向目标基底 `C3`，最后以此将之前另存为临时文件的修改依序应用。（译注：写明了 commit id，以便理解，下同）

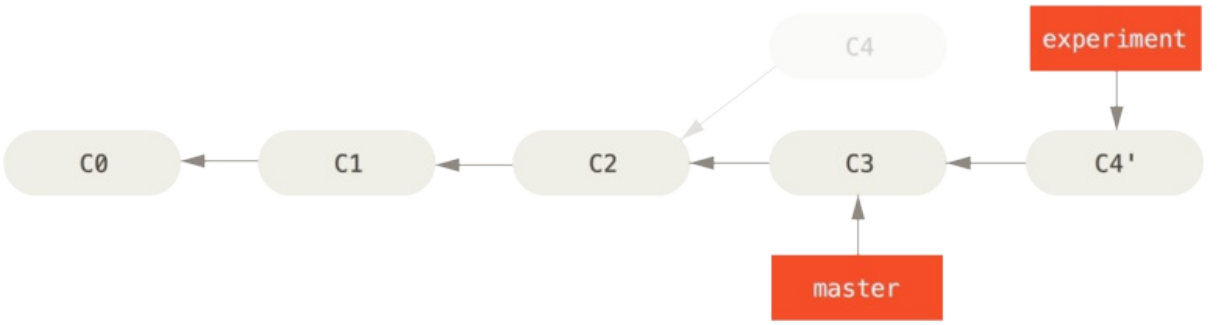


Figure 3-29. 将 `C4` 中的修改变基到 `C3` 上

现在回到 `master` 分支，进行一次快进合并。

```
$ git checkout master
$ git merge experiment
```

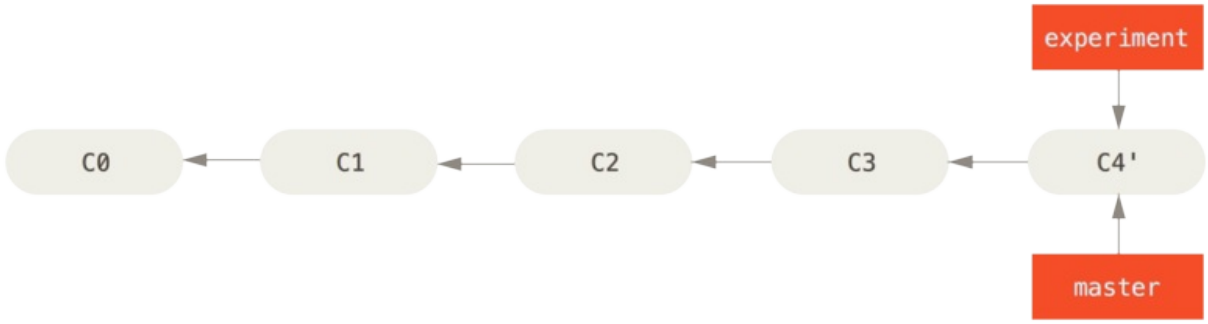


Figure 3-30. master 分支的快进合并

此时，`C4'` 指向的快照就和上面使用 `merge` 命令的例子中 `C5` 指向的快照一模一样了。这两种整合方法的最终结果没有任何区别，但是变基使得提交历史更加整洁。你在查看一个经过变基的分支的历史记录时会发现，尽管实际的开发工作是并行的，但它们看上去就像是先后串行的一样，提交历史是一条直线没有分叉。

一般我们这样做的目的是为了确保在向远程分支推送时能保持提交历史的整洁——例如向某个别人维护的项目贡献代码时。在这种情况下，你首先在自己的分支里进行开发，当开发完成时你需要先将你的代码变基到 `origin/master` 上，然后再向主项目提交修改。这样的话，该项目的维护者就不再需要进行整合工作，只需要快进合并便可。

请注意，无论是通过变基，还是通过三方合并，整合的最终结果所指向的快照始终是一样的，只不过提交历史不同罢了。变基是将一系列提交按照原有次序依次应用到另一分支上，而合并是把最终结果合在一起。

更有趣的变基例子

在对两个分支进行变基时，所生成的“重演”并不一定要在目标分支上应用，你也可以指定另外的一个分支进行应用。就像 Figure 3-31 中的例子这样。你创建了一个特性分支 `server`，为服务端添加了一些功能，提交了 `C3` 和 `C4`。然后从 `C3` 上创建了特性分支 `client`，为客户端添加了一些功能，提交了 `C8` 和 `C9`。最后，你回到 `server` 分支，又提交了 `C10`。

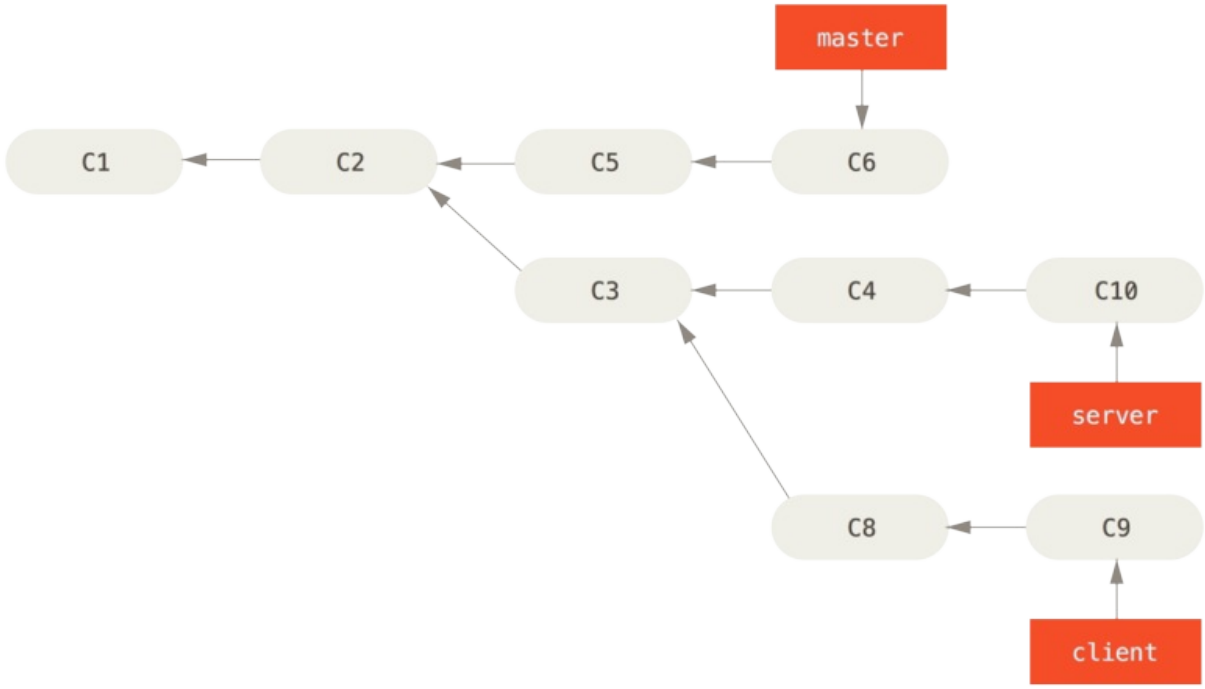


Figure 3-31. 从一个特性分支里再分出一个特性分支的提交历史

假设你希望将 `client` 中的修改合并到主分支并发布，但暂时并不想合并 `server` 中的修改，因为它们还需要经过更全面的测试。这时，你就可以使用 `git rebase` 命令的 `--onto` 选项，选中在 `client` 分支里但不在 `server` 分支里的修改（即 `C8` 和 `C9`），将它们在 `master` 分支上重演：

```
$ git rebase --onto master server client
```

以上命令的意思是：“取出 `client` 分支，找出处于 `client` 分支和 `server` 分支的共同祖先之后的修改，然后把它们在 `master` 分支上重演一遍”。这理解起来有一点复杂，不过效果非常酷。

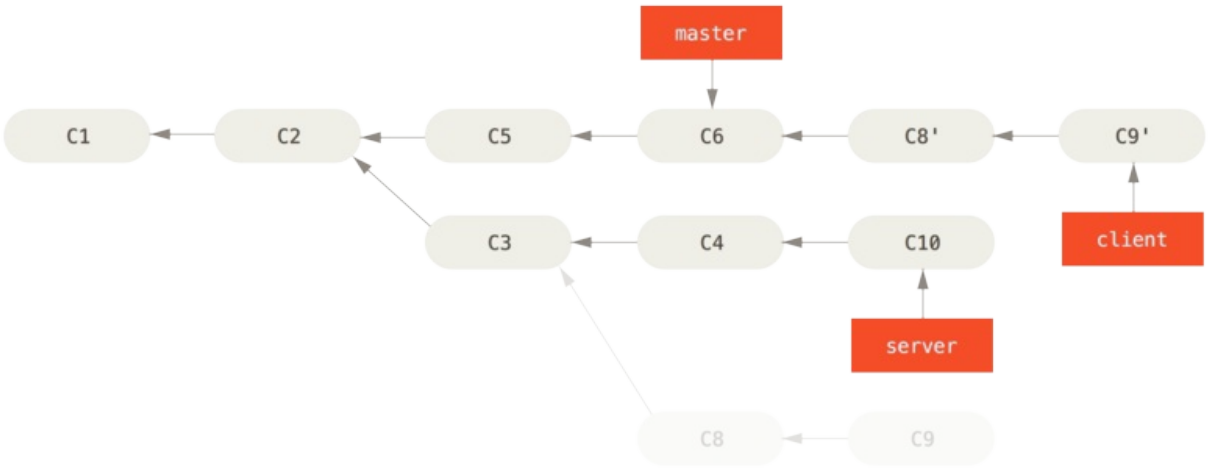


Figure 3-32. 截取特性分支上的另一个特性分支，然后变基到其他分支

现在可以快进合并 `master` 分支了。（如图 Figure 3-33）：

```
$ git checkout master
$ git merge client
```

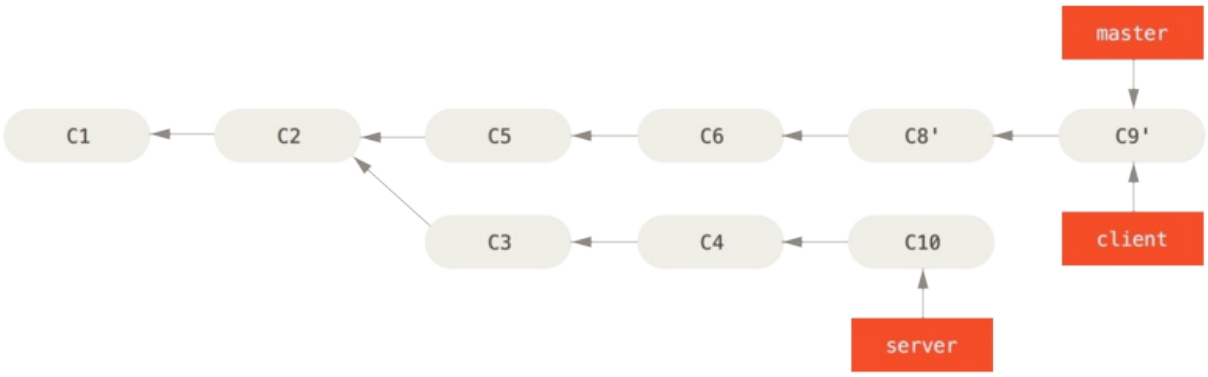


Figure 3-33. 快进合并 master 分支，使之包含来自 client 分支的修改

接下来你决定将 `server` 分支中的修改也整合进来。使用 `git rebase [basebranch] [topicbranch]` 命令可以直接将特性分支（即本例中的 `server`）变基到目标分支（即 `master`）上。这样做能省去你先切换到 `server` 分支，再对其执行变基命令的多个步骤。

```
$ git rebase master server
```

如图 Figure 3-34 所示，`server` 中的代码被“续”到了 `master` 后面。

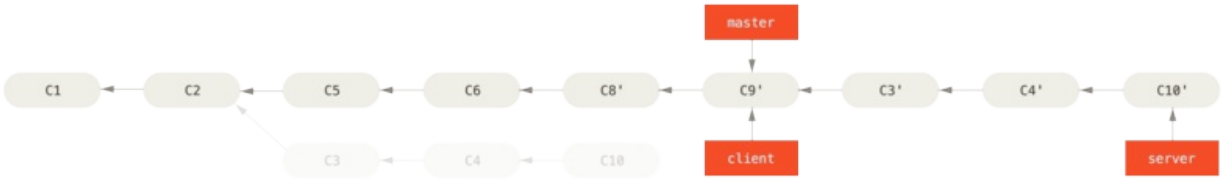


Figure 3-34. 将 server 中的修改变基到 master 上

然后就可以快进合并主分支 master 了：

```
$ git checkout master
$ git merge server
```

至此，`client` 和 `server` 分支中的修改都已经整合到主分支里去了，你可以删除这两个分支，最终提交历史会变成图 Figure 3-35 中的样子：

```
$ git branch -d client
$ git branch -d server
```



Figure 3-35. 最终的提交历史

变基的风险

呃，奇妙的变基也并非完美无缺，要用它得遵守一条准则：

不要对在你的仓库外有副本的分支执行变基。

如果你遵循这条金科玉律，就不会出差错。否则，人民群众会仇恨你，你的朋友和家人也会嘲笑你，唾弃你。

变基操作的实质是丢弃一些现有的提交，然后相应地新建一些内容一样但实际上不同的提交。如果你已经将提交推送至某个仓库，而其他人也已经从该仓库拉取提交并进行了后续工作，此时，如果你用 `git rebase` 命令重新整理了提交并再次推送，你的同伴因此将不得不再次将他们手头的工作与你的提交进行整合，如果接下来你还要拉取并整合他们修改过的提交，事情就会变得一团糟。

让我们来看一个在公开的仓库上执行变基操作所带来的问题。假设你从一个中央服务器克隆然后在它的基础上进行了一些开发。你的提交历史如图所示：

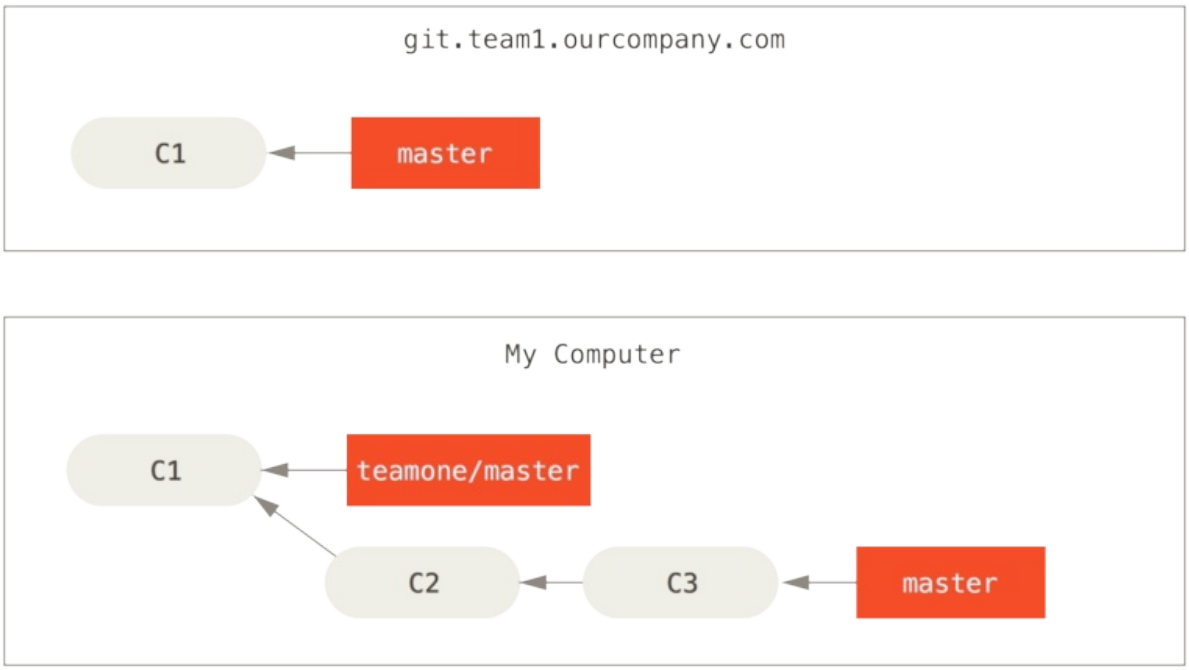


Figure 3-36. 克隆一个仓库，然后在它的基础上进行了一些开发

然后，某人又向中央服务器提交了一些修改，其中还包括一次合并。你抓取了这些在远程分支上的修改，并将其合并到你本地的开发分支，然后你的提交历史就会变成这样：

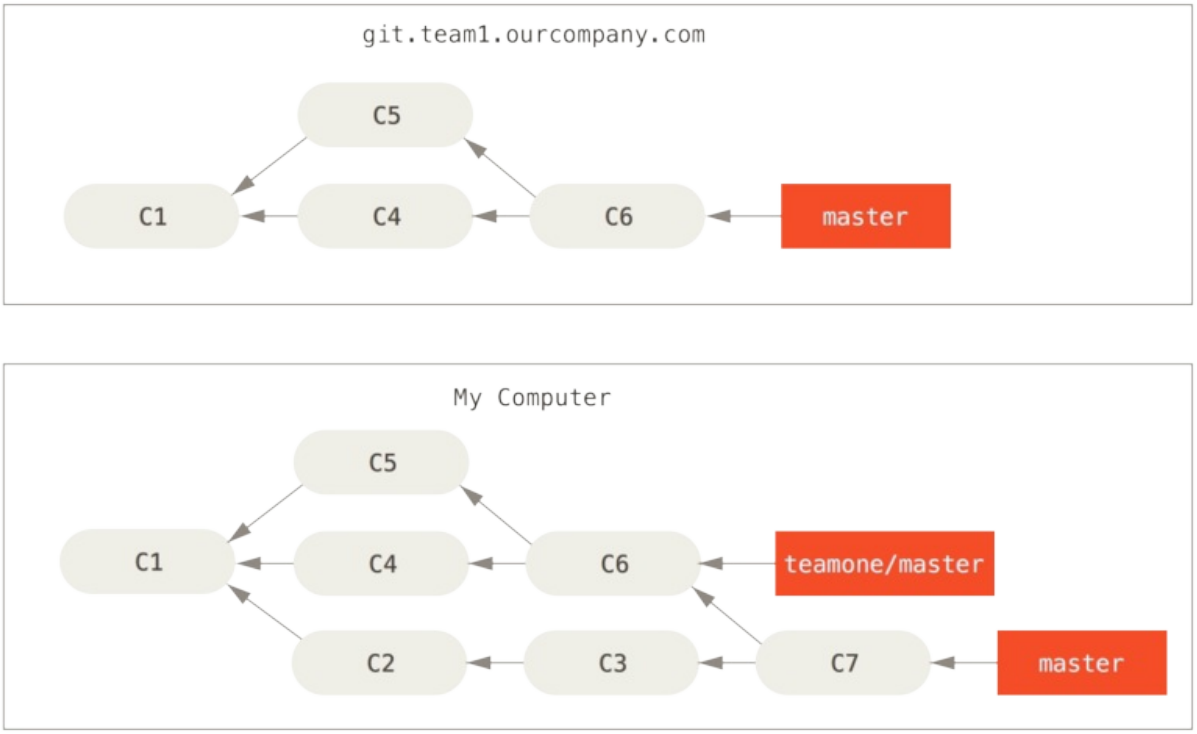


Figure 3-37. 抓取别人的提交，合并到自己的开发分支

接下来，这个人又决定把合并操作回滚，改用变基；继而又用 `git push --force` 命令覆盖了服务器上的提交历史。之后你从服务器抓取更新，会发现多出来一些新的提交。

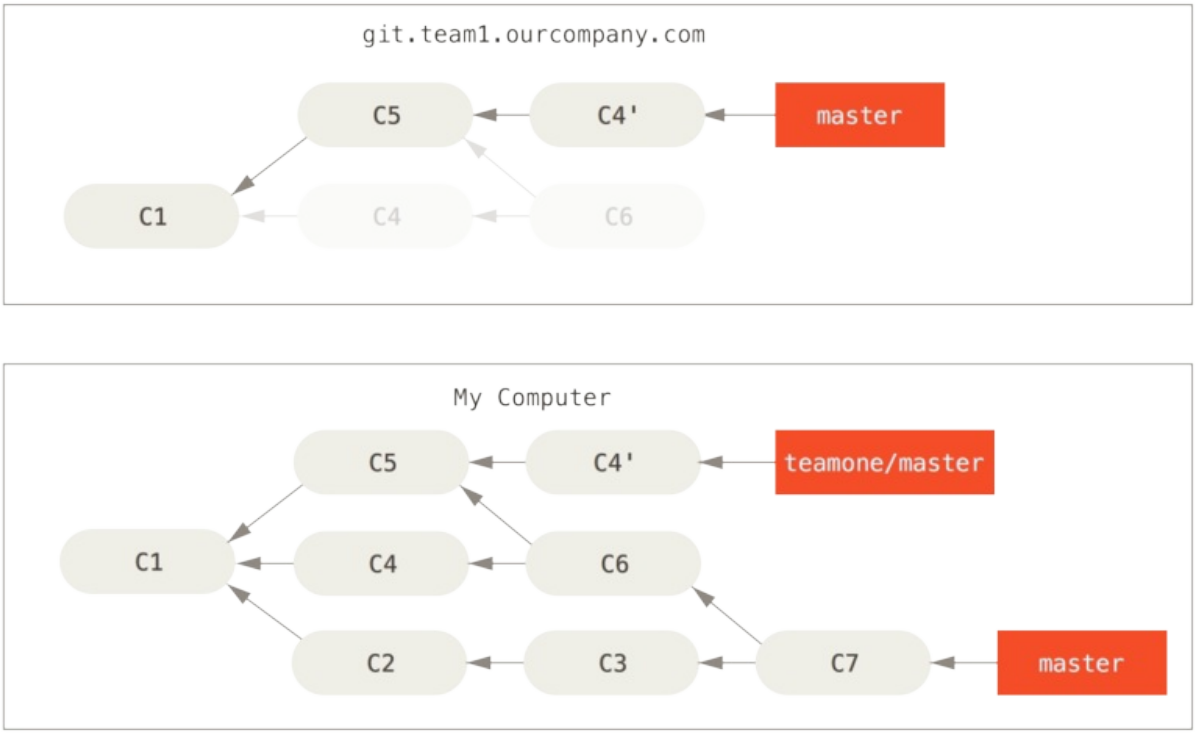


Figure 3-38. 有人推送了经过变基的提交，并丢弃了你的本地开发所基于的一些提交

结果就是你们两人的处境都十分尴尬。如果你执行 `git pull` 命令，你将合并来自两条提交历史的内容，生成一个新的合并提交，最终仓库会如图所示：

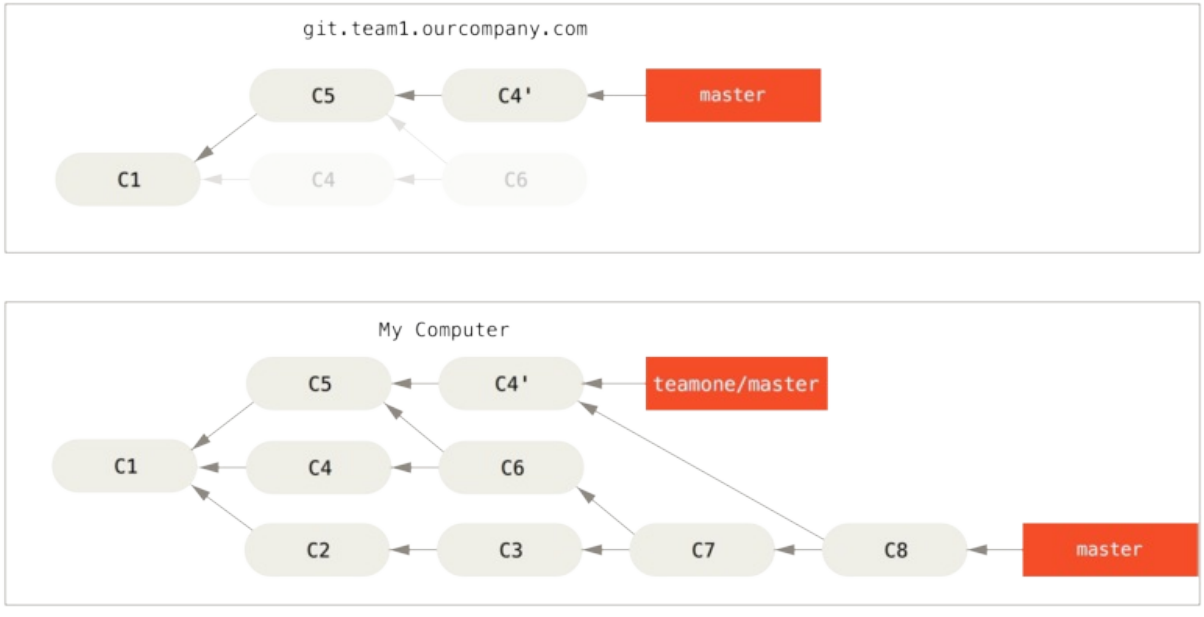


Figure 3-39. 你将相同的内容又合并了一次，生成了一个新的提交

此时如果你执行 `git log` 命令，你会发现有两个提交的作者、日期、日志居然是一样的，这会令人感到混乱。此外，如果你将这一堆又推送到服务器上，你实际上是将那些已经被变基抛弃的提交又找了回来，这会令人感到更加混乱。很明显对方并不想在提交历史中看到 `C4` 和 `C6`，因为之前就是他们把这两个提交通过变基丢弃的。

用变基解决变基

如果你 真的 遭遇了类似的处境，Git 还有一些高级魔法可以帮到你。如果团队中的某人强制推送并覆盖了一些你所基于的提交，你需要做的就是检查你做了哪些修改，以及他们覆盖了哪些修改。

实际上，Git 除了对整个提交计算 SHA-1 校验和以外，也对本次提交所引入的修改计算了校验和——即“patch-id”。

如果你拉取被覆盖过的更新并将你手头的工作基于此进行变基的话，一般情况下 Git 都能成功分辨出哪些是你的修改，并把它们应用到新分支上。

举个例子，如果遇到前面提到的 Figure 3-38 那种情境，如果我们不是执行合并，而是执行 `git rebase teamone/master`，Git 将会：

- 检查哪些提交是我们的分支上独有的（C2，C3，C4，C6，C7）
- 检查其中哪些提交不是合并操作的结果（C2，C3，C4）
- 检查哪些提交在对方覆盖更新时并没有被纳入目标分支（只有 C2 和 C3，因为 C4 其实就是 C4'）
- 把查到的这些提交应用在 `teamone/master` 上面

从而我们将得到与 Figure 3-39 中不同的结果，如图 Figure 3-40 所示。

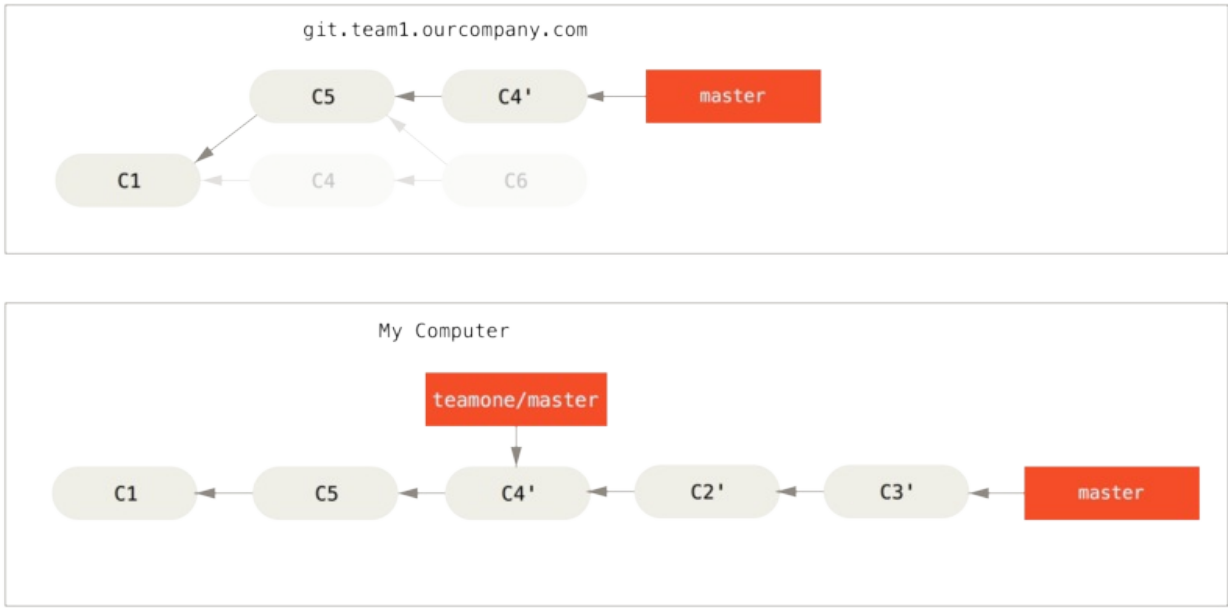


Figure 3-40. 在一个被变基然后强制推送的分支上再次执行变基

要想上述方案有效，还需要对方在变基时确保 C4′ 和 C4 是几乎一样的。否则变基操作将无法识别，并新建另一个类似 C4 的补丁（而这个补丁很可能无法整洁的整合入历史，因为补丁中的修改已经存在于某个地方了）。

在本例中另一种简单的方法是使用 `git pull --rebase` 命令而不是直接 `git pull`。又或者你可以自己手动完成这个过程，先 `git fetch`，再 `git rebase teamone/master`。

如果你习惯使用 `git pull`，同时又希望默认使用选项 `--rebase`，你可以执行这条语句 `git config --global pull.rebase true` 来更改 `pull.rebase` 的默认配置。

只要你把变基命令当作是在推送前清理提交使之整洁的工具，并且只在从未推送至共用仓库的提交上执行变基命令，你就不会有事。假如你在那些已经被推送至共用仓库的提交上执行变基命令，并因此丢弃了一些别人的开发所基于的提交，那你就有大麻烦了，你的同事也会因此鄙视你。

如果你或你的同事在某些情形下决意要这么做，请一定要通知每个人执行 `git pull --rebase` 命令，这样尽管不能避免伤痛，但能有所缓解。

变基 vs. 合并

至此，你已在实战中学习了变基和合并的用法，你一定会想问，到底哪种方式更好。在回答这个问题之前，让我们退后一步，想讨论一下提交历史到底意味着什么。

有一种观点认为，仓库的提交历史即是 记录实际发生过什么。它是针对历史的文档，本身就有价值，不能乱改。从这个角度看来，改变提交历史是一种亵渎，你使用谎言掩盖了实际发生过的事情。如果由合并产生的提交历史是一团糟怎么办？既然事实就是如此，那么这些痕迹就应该被保留下来，让后人能够查阅。

另一种观点则正好相反，他们认为提交历史是 项目过程中发生的故事。没人会出版一本书的第一批草稿，软件维护手册也是需要反复修订才能方便使用。持这一观点的人会使用 rebase 及 filter-branch 等工具来编写故事，怎么方便后来的读者就怎么写。

现在，让我们回到之前的问题上来，到底合并还是变基好？希望你能明白，并没有一个简单的答案。Git 是一个非常强大的工具，它允许你对提交历史做许多事情，但每个团队、每个项目对此的需求并不相同。既然你已经分别学习了两者的用法，相信你能够根据实际情况作出明智的选择。

总的原则是，只对尚未推送或分享给别人的本地修改执行变基操作清理历史，从不对已推送至别处的提交执行变基操作，这样，你才能享受到两种方式带来的便利。

3.7 总结

总结

我们已经讲完了 `Git` 分支与合并的基础知识。你现在应该能自如地创建并切换至新分支、在不同分支之间切换以及合并本地分支。你现在应该也能通过推送你的分支至共享服务以分享它们、使用共享分支与他人协作以及在共享之前使用变基操作合并你的分支。下一章，我们将要讲到，如果你想要运行自己的 Git 仓库托管服务器，你需要知道些什么。

4. 服务器上的 Git

第4章. 服务器上的 Git

到目前为止，你应该已经有办法使用 Git 来完成日常工作。然而，为了使用 Git 协作功能，你还需要有远程的 Git 仓库。尽管在技术上你可以从个人仓库进行推送（push）和拉取（pull）来修改内容，但不鼓励使用这种方法，因为一不留心就很容易弄混其他人的进度。此外，你希望你的合作者们即使在你的电脑未联机时亦能存取仓库 — 拥有一个更可靠的公用仓库十分有用。因此，与他人合作的最佳方法即是建立一个你与合作者们都有权利访问，且可从那里推送和拉取资料的共用仓库。

架设一台 Git 服务器并不难。首先，选择你希望服务器使用的通讯协议。在本章第一节将介绍可用的协议以及各自优缺点。下面一节将解释使用那些协议的典型设置及如何在你的服务器上运行。最后，如果你不介意托管你的代码在其他人的服务器，且不想经历设置与维护自己服务器的麻烦，可以试试我们介绍的几个仓库托管服务。

如果你对架设自己的服务器没兴趣，可以跳到本章最后一节去看看如何申请一个代码托管服务的帐户然后继续下一章，我们会在那里讨论分散式源码控制环境的林林总总。

一个远程仓库通常只是一个裸仓库（bare repository）— 即一个没有当前工作目录的仓库。因为该仓库仅仅作为合作媒介，不需要从磁碟检查快照；存放的只有 Git 的资料。简单的说，裸仓库就是你专案目录内的 `.git` 子目录内容，不包含其他资料。

4.1 协议

协议

Git 可以使用四种主要的协议来传输资料：本地协议（Local），HTTP 协议，SSH（Secure Shell）协议及 Git 协议。在此，我们将会讨论那些协议及哪些情形应该使用（或避免使用）他们。

本地协议

最基本的就是 本地协议（Local protocol），其中的远程版本库就是硬盘内的另一个目录。这常见于团队每一个成员都对一个共享的文件系统（例如一个挂载的 NFS）拥有访问权，或者比较少见的多人共用同一台电脑的情况。后者并不理想，因为你的所有代码版本库如果长存于同一台电脑，更可能发生灾难性的损失。

如果你使用共享文件系统，就可以从本地版本库克隆（clone）、推送（push）以及拉取（pull）。像这样去克隆一个版本库或者增加一个远程到现有的项目中，使用版本库路径作为 URL。例如，克隆一个本地版本库，可以执行如下的命令：

```
$ git clone /opt/git/project.git
```

或你可以执行这个命令：

```
$ git clone file:///opt/git/project.git
```

如果在 URL 开头明确的指定 `file://`，那么 Git 的行为会略有不同。如果仅是指定路径，Git 会尝试使用硬链接（hard link）或直接复制所需要的文件。如果指定 `file://`，Git 会触发平时用于网路传输资料的进程，那通常是传输效率较低的方法。指定 `file://` 的主要目的是取得一个没有外部参考（extraneous references）或对象（object）的干净版本库副本– 通常是在从其他版本控制系统导入后或一些类似情况（参见 Chapter 10 for maintenance tasks）需要这么做。在此我们将使用普通路径，因为这样通常更快。

要增加一个本地版本库到现有的 Git 项目，可以执行如下的命令：

```
$ git remote add local_proj /opt/git/project.git
```

然后，就可以像在网络上一样从远端版本库推送和拉取更新了。

优点

基于文件系统的版本库的优点是简单，并且直接使用了现有的文件权限和网络访问权限。如果你的团队已经有共享文件系统，建立版本库会十分容易。只需要像设置其他共享目录一样，把一个裸版本库的副本放

到大家都可以访问的路径，并设置好读/写的权限，就可以了，我们会在 “在服务器上搭建 Git” 讨论如何导出一个裸版本库。

这也是快速从别人的工作目录中拉取更新的方法。如果你和别人一起合作一个项目，他想让你从版本库中拉取更新时，运行类似 `git pull /home/john/project` 的命令比推送到服务再取回简单多了。

缺点

这种方法的缺点是，通常共享文件系统比较难配置，并且比起基本的网络连接访问，这不方便从多个位置访问。如果你想从家里推送内容，必须先挂载一个远程磁盘，相比网络连接的访问方式，配置不方便，速度也慢。

值得一提的是，如果你使用的是类似于共享挂载的文件系统时，这个方法不一定是最快的。访问本地版本库的速度与你访问数据的速度是一样的。在同一个服务器上，如果允许 Git 访问本地硬盘，一般的通过 NFS 访问版本库要比通过 SSH 访问慢。

最终，这个协议并不保护仓库避免意外的损坏。每一个用户都有“远程”目录的完整 shell 权限，没有方法可以阻止他们修改或删除 Git 内部文件和损坏仓库。

HTTP 协议

Git 通过 HTTP 通信有两种模式。在 Git 1.6.6 版本之前只有一个方式可用，十分简单并且通常是只读模式的。Git 1.6.6 版本引入了一种新的、更智能的协议，让 Git 可以像通过 SSH 那样智能的协商和传输数据。之后几年，这个新的 HTTP 协议因为其简单、智能变的十分流行。新版本的 HTTP 协议一般被称为“智能” HTTP 协议，旧版本的一般被称为“哑” HTTP 协议。我们先了解一下新的“智能” HTTP 协议。

智能（Smart） HTTP 协议

“智能” HTTP 协议的运行方式和 SSH 及 Git 协议类似，只是运行在标准的 HTTP/S 端口上并且可以使用各种 HTTP 验证机制，这意味着使用起来会比 SSH 协议简单的多，比如可以使用 HTTP 协议的用户名 / 密码的基础授权，免去设置 SSH 公钥。

智能 HTTP 协议或许已经是最流行的使用 Git 的方式了，它即支持像 `git://` 协议一样设置匿名服务，也可以像 SSH 协议一样提供传输时的授权和加密。而且只用一个 URL 就可以都做到，省去了为不同的需求设置不同的 URL。如果你要推送到一个需要授权的服务器上（一般来讲都需要），服务器会提示你输入用户名和密码。从服务器获取数据时也一样。

事实上，类似 GitHub 的服务，你在网页上看到的 URL （比如，`https://github.com/schacon/simplegit[]`），和你在克隆、推送（如果你有权限）时使用的是一样的。

哑（Dumb） HTTP 协议

如果服务器没有提供智能 HTTP 协议的服务，Git 客户端会尝试使用更简单的“哑” HTTP 协议。哑 HTTP 协议里 web 服务器仅把裸版本库当作普通文件来对待，提供文件服务。哑 HTTP 协议的优美之处

在于设置起来简单。基本上，只需要把一个裸版本库放在 HTTP 跟目录，设置一个叫做 `post-update` 的挂钩就可以了（见 “Git 钩子” ）。此时，只要能访问 web 服务器上你的版本库，就可以克隆你的版本库。下面是设置从 HTTP 访问版本库的方法：

```
$ cd /var/www/htdocs/
$ git clone --bare /path/to/git_project gitproject.git
$ cd gitproject.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

这样就可以了。Git 自带的 `post-update` 挂钩会默认执行合适的命令（`git update-server-info`），来确保通过 HTTP 的获取和克隆操作正常工作。这条命令会在你通过 SSH 向版本库推送之后被执行；然后别人就可以通过类似下面的命令来克隆：

```
$ git clone https://example.com/gitproject.git
```

这里我们用了 Apache 里设置了常用的路径 `/var/www/htdocs`，不过你可以使用任何静态 web 服务器 —— 只需要把裸版本库放到正确的目录下就可以。Git 的数据是以基本的静态文件形式提供的（详情见 Chapter 10）。

通常的，会在可以提供读 / 写的智能 HTTP 服务和简单的只读的哑 HTTP 服务之间选一个。极少会将二者混合提供服务。

优点

我们将只关注智能 HTTP 协议的优点。

不同的访问方式只需要一个 URL 以及服务器只在需要授权时提示输入授权信息，这两个简便性让终端用户使用 Git 变得非常简单。相比 SSH 协议，可以使用用户名 / 密码授权是一个很大的优势，这样用户就不必须在使用 Git 之前先在本地生成 SSH 密钥对再把公钥上传到服务器。对非资深的使用者，或者系统上缺少 SSH 相关程序的使用者，HTTP 协议的可用性是主要的优势。与 SSH 协议类似，HTTP 协议也非常快和高效。

你也可以在 HTTPS 协议上提供只读版本库的服务，如此你在传输数据的时候就可以加密数据；或者，你甚至可以让客户端使用指定的 SSL 证书。

另一个好处是 HTTP/S 协议被广泛使用，一般的企业防火墙都会允许这些端口的数据通过。

缺点

在一些服务器上，架设 HTTP/S 协议的服务端会比 SSH 协议的棘手一些。除了这一点，用其他协议提供 Git 服务与 “智能” HTTP 协议相比就几乎没有优势了。

如果你在 HTTP 上使用需授权的推送，管理凭证会比使用 SSH 密钥认证麻烦一些。然而，你可以选择使用凭证存储工具，比如 OSX 的 Keychain 或者 Windows 的凭证管理器。参考 “凭证存储” 如何安全地

保存 HTTP 密码。

SSH 协议

架设 Git 服务器时常用 SSH 协议作为传输协议。因为大多数环境下已经支持通过 SSH 访问 —— 即时没有也比较很容易架设。SSH 协议也是一个验证授权的网络协议；并且，因为其普遍性，架设和使用都很容易。

通过 SSH 协议克隆版本库，你可以指定一个 `ssh://` 的 URL：

```
$ git clone ssh://user@server/project.git
```

或者使用一个简短的 scp 式的写法：

```
$ git clone user@server:project.git
```

你也可以不指定用户，Git 会使用当前登录的用户名。

优势

用 SSH 协议的优势有很多。首先，SSH 架设相对简单 —— SSH 守护进程很常见，多数管理员都有使用经验，并且多数操作系统都包含了它及相关的管理工具。其次，通过 SSH 访问是安全的 —— 所有传输数据都要经过授权和加密。最后，与 HTTP/S 协议、Git 协议及本地协议一样，SSH 协议很高效，在传输前也会尽量压缩数据。

缺点

SSH 协议的缺点在于你不能通过他实现匿名访问。即便只要读取数据，使用者也要有通过 SSH 访问你的主机的权限，这使得 SSH 协议不利于开源的项目。如果你只在公司网络使用，SSH 协议可能是你唯一要用到的协议。如果你要同时提供匿名只读访问和 SSH 协议，那么你除了为自己推送架设 SSH 服务以外，还得架设一个可以让其他人访问的服务。

Git 协议

接下来是 Git 协议。这是包含在 Git 里的一个特殊的守护进程；它监听在一个特定的端口（9418），类似于 SSH 服务，但是访问无需任何授权。要让版本库支持 Git 协议，需要先创建一个 `git-daemon-export-ok` 文件 —— 它是 Git 协议守护进程为这个版本库提供服务的必要条件 —— 但是除此之外没有任何安全措施。要么谁都可以克隆这个版本库，要么谁也不能。这意味这，通常不能通过 Git 协议推送。由于没有授权机制，一旦你开放推送操作，意味着网络上知道这个项目 URL 的人都可以向项目推送数据。不用说，极少会有人这么做。

优点

目前，Git 协议是 Git 使用的网络传输协议里最快的。如果你的项目有很大的访问量，或者你的项目很庞大并且不需要为写进行用户授权，架设 Git 守护进程来提供服务是不错的选择。它使用与 SSH 相同的数

据传输机制，但是省去了加密和授权的开销。

缺点

Git 协议缺点是缺乏授权机制。把 Git 协议作为访问项目版本库的唯一手段是不可取的。一般的做法里，会同时提供 SSH 或者 HTTPS 协议的访问服务，只让少数几个开发者有推送（写）权限，其他人通过 `git://` 访问只有读权限。Git 协议也许也是最难架设的。它要求有自己的守护进程，这就要配置 `xinetd` 或者其他的程序，这些工作并不简单。它还要求防火墙开放 9418 端口，但是企业防火墙一般不会开放这个非标准端口。而大型的企业防火墙通常会封锁这个端口。

4.2 在服务器上搭建 Git

在服务器上搭建 Git

现在我们将讨论如何在你自己的服务器上搭建 Git 服务来运行这些协议。

> 这里我们将要演示在 Linux 服务器上进行一次基本且简化的安装所需的命令与步骤，当然在 Mac 或 Windows 服务器上同样可以运行这些服务。事实上，在你的计算机基础架构中建立一个生产环境服务器，将不可避免的使用到不同的安全措施与操作系统工具。但是，希望你能从本节中获得一些必要的知识。

在开始架设 Git 服务器前，需要把现有仓库导出为裸仓库——即一个不包含当前工作目录的仓库。这通常是很简单的。为了通过克隆你的仓库来创建一个新的裸仓库，你需要在克隆命令后加上 `--bare` 选项按照惯例，裸仓库目录名以 .git 结尾，就像这样：

```
$ git clone --bare my_project my_project.git
Cloning into bare repository 'my_project.git'...
done.
```

现在，你的 `my_project.git` 目录中应该有 Git 目录的副本了。

整体上效果大致相当于

```
$ cp -Rf my_project/.git my_project.git
```

虽然在配置文件中有若干不同，但是对于你的目的来说，这两种方式都是一样的。它只取出 Git 仓库自身，不要工作目录，然后特别为它单独创建一个目录。

把裸仓库放到服务器上

既然你有了裸仓库的副本，剩下要做的就是把裸仓库放到服务器上并设置你的协议。假设一个域名为 `git.example.com` 的服务器已经架设好，并可以通过 SSH 连接，你想把所有的 Git 仓库放在 `/opt/git` 目录下。假设服务器上存在 `/opt/git/` 目录，你可以通过以下命令复制你的裸仓库来创建一个新仓库：

```
$ scp -r my_project.git user@git.example.com:/opt/git
```

此时，其他通过 SSH 连接这台服务器并对 `/opt/git` 目录拥有可读权限的使用者，通过运行以下命令就可以克隆你的仓库。

```
$ git clone user@git.example.com:/opt/git/my_project.git
```

如果一个用户，通过使用 SSH 连接到一个服务器，并且其对 `/opt/git/my_project.git` 目录拥有可写权限，那么他将自动拥有推送权限。

如果到该项目目录中运行 `git init` 命令，并加上 `--shared` 选项，那么 Git 会自动修改该仓库目录的组权限为可写。

```
$ ssh user@git.example.com
$ cd /opt/git/my_project.git
$ git init --bare --shared
```

由此可见，根据现有的 Git 仓库创建一个裸仓库，然后把它放上你和协作者都有 SSH 访问权的服务器是多么容易。现在你们已经准备好在同一项目上展开合作了。

值得注意的是，这的确是架设一个几个人拥有连接权的 Git 服务的全部——只要在服务器上加入可以用 SSH 登录的帐号，然后把裸仓库放在大家都有读写权限的地方。你已经准备好了一切，无需更多。

下面的几节中，你会了解如何扩展到更复杂的设定。这些内容包含如何避免为每一个用户建立一个账户，给仓库添加公共读取权限，架设网页界面等等。然而，请记住这一点，如果只是和几个人在一个私有项目上合作的话，仅仅 是一个 SSH 服务器和裸仓库就足够了。

小型安装

如果设备较少或者你只想在小型开发团队里尝试 Git ，那么一切都很简单。架设 Git 服务最复杂的地方在于用户管理。如果需要仓库对特定的用户可读，而给另一部分用户读写权限，那么访问和许可安排就会比较困难。

SSH 连接

如果你有一台所有开发者都可以用 SSH 连接的服务器，架设你的第一个仓库就十分简单了，因为你几乎什么都不用做（正如我们上一节所说的）。如果你想在你的仓库上设置更复杂的访问控制权限，只要使用服务器操作系统的普通的文件系统权限就行了。

如果需要团队里的每个人都对仓库有写权限，又不能给每个人在服务器上建立账户，那么提供 SSH 连接就是唯一的选择了。我们假设用来共享仓库的服务器已经安装了 SSH 服务，而且你通过它访问服务器。

有几个方法可以使你给团队每个成员提供访问权。第一个就是给团队里的每个人创建账号，这种方法很直接但也很麻烦。或许你不会想要为每个人运行一次 `adduser` 并且设置临时密码。

第二个办法是在主机上建立一个 git 账户，让每个需要写权限的人发送一个 SSH 公钥，然后将其加入 git 账户的 `~/.ssh/authorized_keys` 文件。这样一来，所有人都将通过 git 账户访问主机。这一点也不会影响提交的数据——访问主机用的身份不会影响提交对象的提交者信息。

另一个办法是让 SSH 服务器通过某个 LDAP 服务，或者其他已经设定好的集中授权机制，来进行授权。只要每个用户可以获得主机的 shell 访问权限，任何 SSH 授权机制你都可视为是有效的。

4.3 生成 SSH 公钥

生成 SSH 公钥

如前所述，许多 Git 服务器都使用 SSH 公钥进行认证。为了向 Git 服务器提供 SSH 公钥，如果某系统用户尚未拥有密钥，必须事先为其生成一份。这个过程在所有操作系统上都是相似的。首先，你需要确认自己是否已经拥有密钥。默认情况下，用户的 SSH 密钥存储在其 `~/.ssh` 目录下。进入该目录并列出其中内容，你便可以快速确认自己是否已拥有密钥：

```
$ cd ~/.ssh
$ ls
authorized_keys2  id_dsa       known_hosts
config            id_dsa.pub
```

我们需要寻找一对以 `id_dsa` 或 `id_rsa` 命名的文件，其中一个带有 `.pub` 扩展名。`.pub` 文件是你的公钥，另一个则是私钥。如果找不到这样的文件（或者根本没有 `.ssh` 目录），你可以通过运行 `ssh-keygen` 程序来创建它们。在 Linux/Mac 系统中，`ssh-keygen` 随 SSH 软件包提供；在 Windows 上，该程序包含于 MSysGit 软件包中。

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/schacon/.ssh/id_rsa):
Created directory '/home/schacon/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/schacon/.ssh/id_rsa.
Your public key has been saved in /home/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3 schacon@mylaptop.local
```

首先 `ssh-keygen` 会确认密钥的存储位置（默认是 `.ssh/id_rsa`），然后它会要求你输入两次密钥口令。如果你不想在使用密钥时输入口令，将其留空即可。

现在，进行了上述操作的用户需要将各自的公钥发送给任意一个 Git 服务器管理员（假设服务器正在使用基于公钥的 SSH 验证设置）。他们所要做的就是复制各自的 `.pub` 文件内容，并将其通过邮件发送。公钥看起来是这样的：

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

关于在多种操作系统中生成 SSH 密钥的更深入教程，请参阅 GitHub 的 SSH 密钥指南 https://help.github.com/articles/generating-ssh-keys。

4.4 配置服务器

配置服务器

我们来看看如何配置服务器端的 SSH 访问。本例中，我们将使用 `authorized_keys` 方法来对用户进行认证。同时我们假设你使用的操作系统是标准的 Linux 发行版，比如 Ubuntu。首先，创建一个操作系统用户 `git`，并为其建立一个 `.ssh` 目录。

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh && chmod 700 .ssh
$ touch .ssh/authorized_keys && chmod 600 .ssh/authorized_keys
```

接着，我们需要为系统用户 `git` 的 `authorized_keys` 文件添加一些开发者 SSH 公钥。假设我们已经获得了若干受信任的公钥，并将它们保存在临时文件中。与前文类似，这些公钥看起来是这样的：

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv
O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

将这些公钥加入系统用户 `git` 的 `.ssh` 目录下 `authorized_keys` 文件的末尾：

```
$ cat /tmp/id_rsa.john.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.josie.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.jessica.pub >> ~/.ssh/authorized_keys
```

现在我们来为开发者新建一个空仓库。可以借助带 `--bare` 选项的 `git init` 命令来做到这一点，该命令在初始化仓库时不会创建工作目录：

```
$ cd /opt/git
$ mkdir project.git
$ cd project.git
$ git init --bare
Initialized empty Git repository in /opt/git/project.git/
```

接着，John、Josie 或者 Jessica 中的任意一人可以将他们项目的最初版本推送到这个仓库中，他只需将此仓库设置为项目的远程仓库并向其推送分支。请注意，每添加一个新项目，都需要有人登录服务器取得 shell，并创建一个裸仓库。我们假定这个设置了 `git` 用户和 Git 仓库的服务器使用 `gitserver` 作为

主机名。同时，假设该服务器运行在内网，并且你已在 DNS 配置中将 `gitserver` 指向此服务器。那么我们可以运行如下命令（假定 `myproject` 是已有项目且其中已包含文件）：

```
# on John's computer
$ cd myproject
$ git init
$ git add .
$ git commit -m 'initial commit'
$ git remote add origin git@gitserver:/opt/git/project.git
$ git push origin master
```

此时，其他开发者可以克隆此仓库，并推回各自的改动，步骤很简单：

```
$ git clone git@gitserver:/opt/git/project.git
$ cd project
$ vim README
$ git commit -am 'fix for the README file'
$ git push origin master
```

通过这种方法，你可以快速搭建一个具有读写权限、面向多个开发者的 Git 服务器。

需要注意的是，目前所有（获得授权的）开发者用户都能以系统用户 `git` 的身份登录服务器从而获得一个普通 shell。如果你想对此加以限制，则需要修改 `passwd` 文件中（ `git` 用户所对应）的 shell 值。

借助一个名为 `git-shell` 的受限 shell 工具，你可以方便地将用户 `git` 的活动限制在与 Git 相关的范围内。该工具随 Git 软件包一同提供。如果将 `git-shell` 设置为用户 `git` 的登录 shell（login shell），那么用户 `git` 便不能获得此服务器的普通 shell 访问权限。若要使用 `git-shell`，需要用它替换掉 bash 或 csh，使其成为系统用户的登录 shell。为进行上述操作，首先你必须确保 `git-shell` 已存在于 `/etc/shells` 文件中：

```
$ cat /etc/shells   # see if `git-shell` is already in there.  If not...
$ which git-shell   # make sure git-shell is installed on your system.
$ sudo vim /etc/shells  # and add the path to git-shell from last command
```

现在你可以使用 `chsh <username>` 命令修改任一系统用户的 shell：

```
$ sudo chsh git  # and enter the path to git-shell, usually: /usr/bin/git-shell
```

这样，用户 `git` 就只能利用 SSH 连接对 Git 仓库进行推送和拉取操作，而不能登录机器并取得普通 shell。如果试图登录，你会发现尝试被拒绝，像这样：

```
$ ssh git@gitserver
fatal: Interactive git shell is not enabled.
hint: ~/git-shell-commands should exist and have read and execute access.
Connection to gitserver closed.
```

现在，网络相关的 Git 命令依然能够正常工作，但是开发者用户已经无法得到一个普通 shell 了。正如输出信息所提示的，你也可以在 `git` 用户的家目录下建立一个目录，来对 `git-shell` 命令进行一定程度的自定义。比如，你可以限制掉某些本应被服务器接受的 Git 命令，或者对刚才的 SSH 拒绝登录信息进行自定义，这样，当有开发者用户以类似方式尝试登录时，便会看到你的信息。要了解更多有关自定义 shell 的信息，请运行 `git help shell`。

4.5 Git 守护进程

Git 守护进程

接下来我们将通过 “Git” 协议建立一个基于守护进程的仓库。对于快速且无需授权的 Git 数据访问，这是一个理想之选。请注意，因为其不包含授权服务，任何通过该协议管理的内容将在其网络上公开。

如果运行在防火墙之外的服务器上，它应该只对那些公开的只读项目服务。如果运行在防火墙之内的服务器上，它可用于支撑大量参与人员或自动系统（用于持续集成或编译的主机）只读访问的项目，这样可以省去逐一配置 SSH 公钥的麻烦。

无论何时，该 Git 协议都是相对容易设定的。通常，你只需要以守护进程的形式运行该命令：

```
git daemon --reuseaddr --base-path=/opt/git/ /opt/git/
```

`--reuseaddr` 允许服务器在无需等待旧连接超时的情况下重启，`--base-path` 选项允许用户在未完全指定路径的条件下克隆项目，结尾的路径将告诉 Git 守护进程从何处寻找仓库来导出。如果有防火墙正在运行，你需要开放端口 9418 的通信权限。

你可以通过许多方式将该进程以守护进程的方式运行，这主要取决于你所使用的操作系统。在一台 Ubuntu 机器上，你可以使用一份 Upstart 脚本。因此，找到如下文件：

```
/etc/event.d/local-git-daemon
```

并添加下列脚本内容：

```
start on startup
stop on shutdown
exec /usr/bin/git daemon \
    --user=git --group=git \
    --reuseaddr \
    --base-path=/opt/git/ \
    /opt/git/
respawn
```

出于安全考虑，强烈建议使用一个对仓库拥有只读权限的用户身份来运行该守护进程 - 你可以创建一个新用户 git-ro 并且以该用户身份来运行守护进程。为简便起见，我们将像 `git-shell` 一样，同样使用 git 用户来运行它。

当你重启机器时，你的 Git 守护进程将会自动启动，并且如果进程被意外结束它会自动重新运行。为了在不重启的情况下直接运行，你可以运行以下命令：

```
initctl start local-git-daemon
```

在其他系统中，你可以使用 `sysvinit` 系统中的 `xinetd` 脚本，或者另外的方式来实现 - 只要你能够将其命令守护进程化并实现监控。

接下来，你需要告诉 Git 哪些仓库允许基于服务器的无授权访问。你可以在每个仓库下创建一个名为 `git-daemon-export-ok` 的文件来实现。

```
$ cd /path/to/project.git
$ touch git-daemon-export-ok
```

该文件将允许 Git 提供无需授权的项目访问服务。

4.6 Smart HTTP

Smart HTTP

我们一般通过 SSH 进行授权访问，通过 git:// 进行无授权访问，但是还有一种协议可以同时实现以上两种方式的访问。设置 Smart HTTP 一般只需要在服务器上启用一个 Git 自带的名为 `git-http-backend` 的 CGI 脚本。该 CGI 脚本将会读取由 `git fetch` 或 `git push` 命令向 HTTP URL 发送的请求路径和头部信息，来判断该客户端是否支持 HTTP 通信（不低于 1.6.6 版本的客户端支持此特性）。如果 CGI 发现该客户端支持智能（Smart）模式，它将会以智能模式与它进行通信，否则它将会回落到哑（Dumb）模式下（因此它可以对某些老的客户端实现向下兼容）。

在完成以上简单的安装步骤后，我们将用 Apache 来作为 CGI 服务器。如果你没有安装 Apache，你可以在 Linux 环境下执行如下或类似的命令来安装：

```
$ sudo apt-get install apache2 apache2-utils
$ a2enmod cgi alias env
```

该操作将会启用 `mod_cgi`， `mod_alias`， 和 `mod_env` 等 Apache 模块， 这些模块都是使该功能正常工作所必须的。

接下来我们要向 Apache 配置文件添加一些内容，来让 `git-http-backend` 作为 Web 服务器对 `/git` 路径请求的处理器。

```
SetEnv GIT_PROJECT_ROOT /opt/git
SetEnv GIT_HTTP_EXPORT_ALL
ScriptAlias /git/ /usr/lib/git-core/git-http-backend/
```

如果留空 `GIT_HTTP_EXPORT_ALL` 这个环境变量，Git 将只对无授权客户端提供带 `git-daemon-export-ok` 文件的版本库，就像 Git 守护进程一样。

接着你需要让 Apache 接受通过该路径的请求，添加如下的内容至 Apache 配置文件：

```
<Directory "/usr/lib/git-core*">
   Options ExecCGI Indexes
   Order allow,deny
   Allow from all
   Require all granted
</Directory>
```

最后，如果想实现写操作授权验证，使用如下的未授权屏蔽配置即可：

```
<LocationMatch "^/git/.*/git-receive-pack$">
    AuthType Basic
    AuthName "Git Access"
    AuthUserFile /opt/git/.htpasswd
    Require valid-user
</LocationMatch>
```

这需要你创建一个包含所有合法用户密码的 `.htaccess` 文件。以下是一个添加 “schacon” 用户到此文件的例子：

```
$ htdigest -c /opt/git/.htpasswd "Git Access" schacon
```

你可以通过许多方式添加 Apache 授权用户，选择使用其中一种方式即可。以上仅仅只是我们可以找到的最简单的一个例子。如果愿意的话，你也可以通过 SSL 运行它，以保证所有数据是在加密状态下进行传输的。

我们不想深入去讲解 Apache 配置文件，因为你可能会使用不同的 Web 服务器，或者可能有不同的授权需求。它的主要原理是使用一个 Git 附带的，名为 `git-http-backend` 的 CGI。它被引用来处理协商通过 HTTP 发送和接收的数据。它本身并不包含任何授权功能，但是授权功能可以在 Web 服务器层引用它时被轻松实现。你可以在任何所有可以处理 CGI 的 Web 服务器上办到这点，所以随便挑一个你最熟悉的 Web 服务器试手吧。

> 欲了解更多的有关配置 Apache 授权访问的信息，请通过以下链接浏览 Apache 文档：
> http://httpd.apache.org/docs/current/howto/auth.html

4.7 GitWeb

GitWeb

如果你对项目有读写权限或只读权限，你可能需要建立起一个基于网页的简易查看器。Git 提供了一个叫做 GitWeb 的 CGI 脚本来做这项工作。

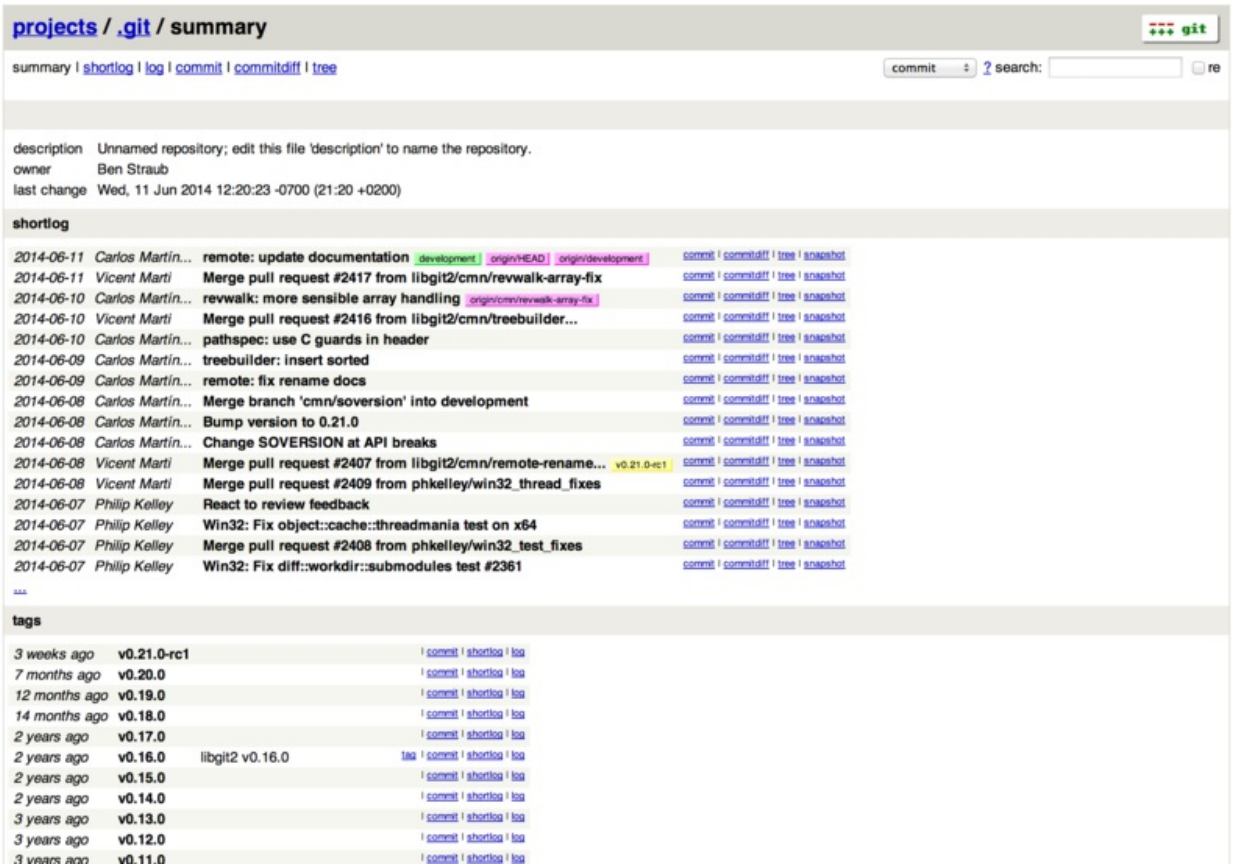


Figure 4-1. GitWeb 的网页用户界面

如果你想要查看 GitWeb 如何展示你的项目，并且在服务器上安装了轻量级网络服务器比如 `lighttpd` 或 `webrick`， Git 提供了一个命令来让你启动一个临时的服务器。在 Linux 系统的电脑上，`lighttpd` 通常已经安装了，所以你只需要在项目目录里执行 `git instaweb` 命令即可。如果你使用 Mac 系统， Mac OS X Leopard 系统已经预安装了 Ruby，所以 `webrick` 或许是你最好的选择。如果不想使用 lighttpd 启动 `instaweb` 命令，你需要在执行时加入 `--httpd` 参数。

```
$ git instaweb --httpd=webrick
[2009-02-21 10:02:21] INFO  WEBrick 1.3.1
[2009-02-21 10:02:21] INFO  ruby 1.8.6 (2008-03-03) [universal-darwin9.0]
```

这个命令启动了一个监听 1234 端口的 HTTP 服务器，并且自动打开了浏览器。这对你来说十分方便。当你已经完成了工作并想关闭这个服务器，你可以执行同一个命令，并加上 `--stop` 选项：

```
$ git instaweb --httpd=webrick --stop
```

如果你现在想为你的团队或你托管的开源项目持续的运行这个页面，你需要通过普通的 Web 服务器来设置 CGI 脚本。一些 Linux 发行版的软件库有 `gitweb` 包，可以通过 `apt` 或 `yum` 来安装，你可以先试试。接下来我们来快速的了解一下如何手动安装 GitWeb。首先，你需要获得 Git 的源代码，它包含了 GitWeb ，并可以生成自定义的 CGI 脚本：

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/opt/git" prefix=/usr gitweb
    SUBDIR gitweb
    SUBDIR ../
make[2]: `GIT-VERSION-FILE' is up to date.
    GEN gitweb.cgi
    GEN static/gitweb.js
$ sudo cp -Rf gitweb /var/www/
```

需要注意的是，你需要在命令中指定 `GITWEB_PROJECTROOT` 变量来让程序知道你的 Git 版本库的位置。现在，你需要在 Apache 中使用这个 CGI 脚本，你需要为此添加一个虚拟主机：

```
<VirtualHost *:80>
    ServerName gitserver
    DocumentRoot /var/www/gitweb
    <Directory /var/www/gitweb>
        Options ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch
        AllowOverride All
        order allow,deny
        Allow from all
        AddHandler cgi-script cgi
        DirectoryIndex gitweb.cgi
    </Directory>
</VirtualHost>
```

再次提醒，GitWeb 可以通过任何一个支持 CGI 或 Perl 的网络服务器架设；如果你需要的话，架设起来应该不会很困难。现在，你可以访问 `http://gitserver/` 在线查看你的版本库。

4.8 GitLab

GitLab

虽然 GitWeb 相当简单。但如果你正在寻找一个更现代，功能更全的 Git 服务器，这里有几个开源的解决方案可供你选择安装。因为 GitLab 是其中最出名的一个，我们将它作为示例并讨论它的安装和使用。这比 GitWeb 要复杂的多并且需要更多的维护，但它的确是一个功能更全的选择。

安装

GitLab 是一个数据库支持的 web 应用，所以相比于其他 git 服务器，它的安装过程涉及到更多的东西。幸运的是，这个过程有非常详细的文档说明和支持。

这里有一些可参考的方法帮你安装 GitLab 。为了更快速的启动和运行，你可以下载虚拟机镜像或者在 https://bitnami.com/stack/gitlab 上获取一键安装包，同时调整配置使之符合你特定的环境。Bitnami 的一个优点在于它的登录界面（通过 alt-&rarr 键进入；）；它会告诉你安装好的 GitLab 的 IP 地址以及默认的用户名和密码。



Figure 4-2. Bitnami GitLab 虚拟机登录界面。

无论如何，跟着 GitLab 社区版的 readme 文件一步步来，你可以在这里找到它 https://gitlab.com/gitlab-org/gitlab-ce/tree/master 。在这里你将会在主菜单中找到安装 GitLab 的帮助，一个可以在 Digital Ocean 上运行的虚拟机，以及 RPM 和 DEB 包（都是测试版）。这里还有“非官方” 的引导让 GitLab 运行在非标准的操作系统和数据库上，一个全手动的安装脚本，以及许多其

他的话题。

管理

GitLab 的管理界面是通过网络进入的。将你的浏览器转到已经安装 GitLab 的 主机名或 IP 地址，然后以管理员身份登录即可。默认的用户名是 `admin@local.host`，默认的密码是 `5iveL!fe`（你会得到类似 请登录后尽快更换密码 的提示）。登录后，点击主栏上方靠右位置的 “Admin area” 图标进行管理。

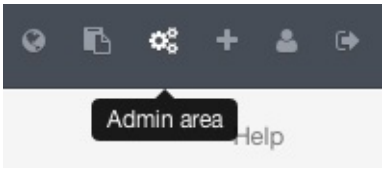


Figure 4-3. GitLab 主栏的 “Admin area” 图标。

使用者

GitLab 上的用户指的是对应协作者的帐号。用户帐号没有很多复杂的地方，主要是包含登录数据的用户信息集合。每一个用户账号都有一个 命名空间 ，即该用户项目的逻辑集合。如果一个叫 `jane` 的用户拥有一个名称是 `project` 的项目，那么这个项目的 url 会是 http://server/jane/project 。

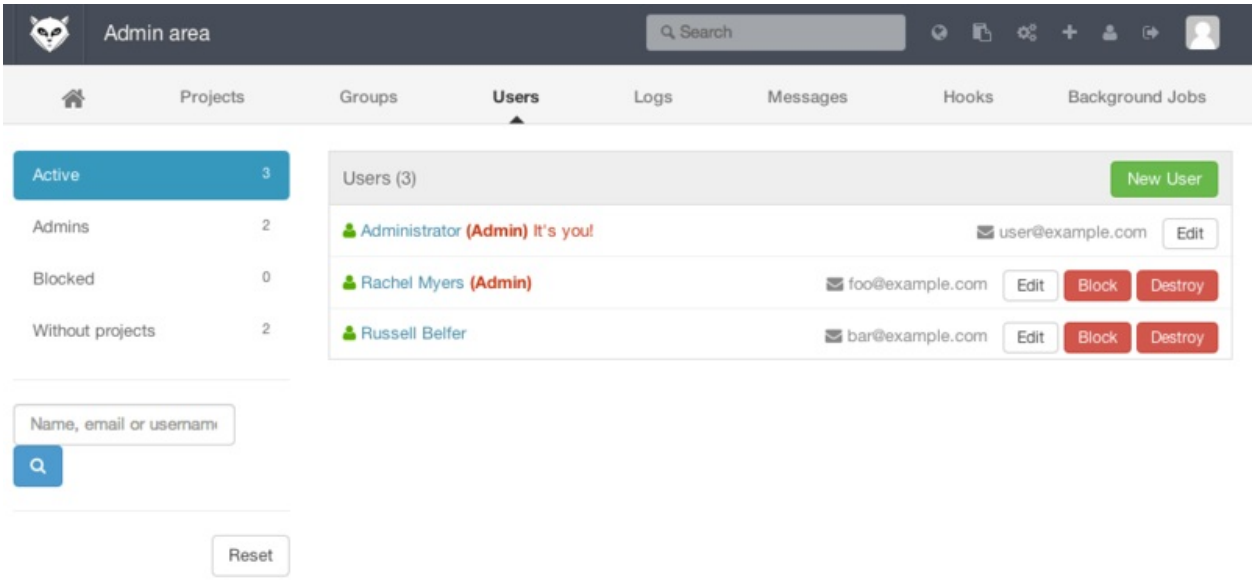


Figure 4-4. GitLab 用户管理界面。

移除一个用户有两种方法。 “屏蔽（Blocking）” 一个用户阻止他登录 GitLab 实例，但是该用户命名空间下的所有数据仍然会被保存，并且仍可以通过该用户提交对应的登录邮箱链接回他的个人信息页。

而另一方面， “销毁（Destroying）” 一个用户，会彻底的将他从数据库和文件系统中移除。他命名空间下的所有项目和数据都会被删除，拥有的任何组也会被移除。这显然是一个更永久且更具破坏力的行为，所以很少用到这种方法。

组

一个 GitLab 的组是一些项目的集合，连同关于多少用户可以访问这些项目的数据。每一个组都有一个项目命名空间（与用户一样），所以如果一个叫 `training` 的组拥有一个名称是 `materials` 的项目，那么这个项目的 url 会是 http://server/training/materials 。

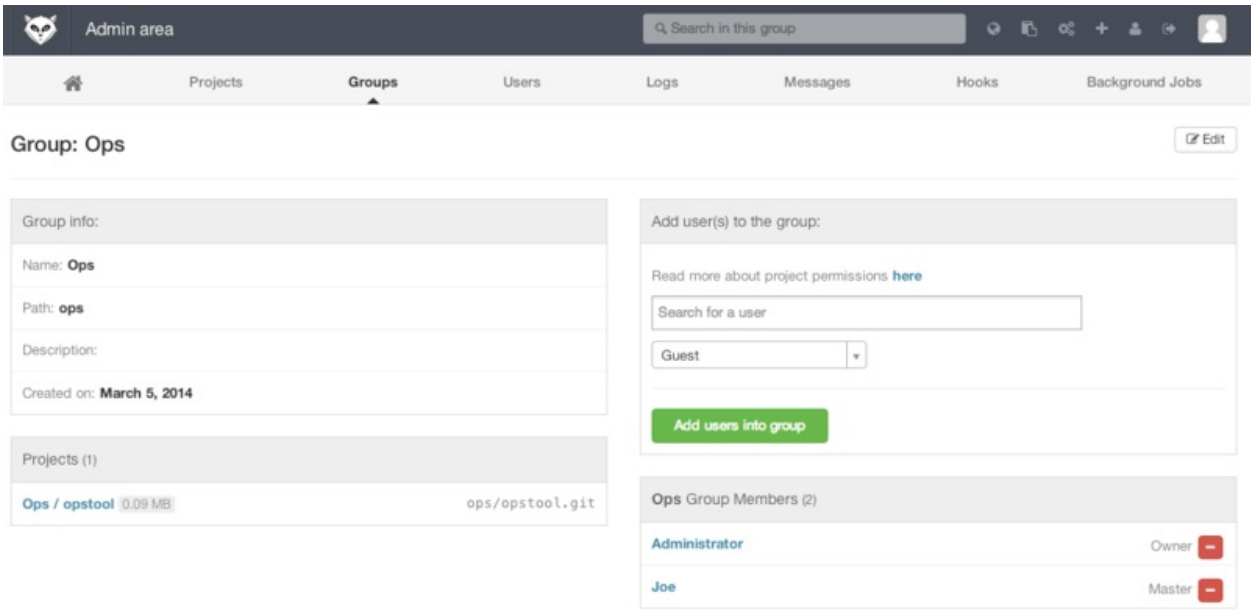


Figure 4-5. GitLab组 管理界面。

每一个组都有许多用户与之关联，每一个用户对组中的项目以及组本身的权限都有级别区分。权限的范围从 “访客”（仅能提问题和讨论）到 “拥有者”（完全控制组、成员和项目）。权限的种类太多以至于难以在这里一一列举，不过在 GitLab 的管理界面上有帮助链接。

项目

一个 GitLab 的项目相当于 git 的版本库。每一个项目都属于一个用户或者一个组的单个命名空间。如果这个项目属于一个用户，那么这个拥有者对所有可以获取这个项目的人拥有直接管理权；如果这个项目属于一个组，那么该组中用户级别的权限也会起作用。

每一个项目都有一个可视级别，控制着谁可以看到这个项目页面和仓库。如果一个项目是 私有 的，这个项目的拥有者必须明确授权从而使特定的用户可以访问。一个 内部 的项目可以被所有登录的人看到，而一个 公开 的项目则是对所有人可见的。注意，这种控制既包括 git “fetch” 的使用也包括对项目 web 用户界面的访问。

钩子

GitLab 在项目和系统级别上都支持钩子程序。对任意级别，当有相关事件发生时，GitLab 的服务器会执行一个包含描述性 JSON 数据的 HTTP 请求。这是自动化连接你的 git 版本库和 GitLab 实例到其他的开发工具，比如 CI 服务器，聊天室，或者部署工具的一个极好方法。

基本用途

你想要在 GitLab 做的第一件事就是建立一个新项目。这通过点击工具栏上的 “+” 图标完成。你会被要求填写项目名称，也就是这个项目所属的命名空间，以及它的可视层级。绝大多数的设定并不是永久的，可以通过设置界面重新调整。点击 “Create Project” ，你就完成了。

项目存在后，你可能会想将它与本地的 Git 版本库连接。每一个项目都可以通过 HTTPS 或者 SSH 连接，任意两者都可以被用来配置远程 Git。在项目主页的顶栏可以看到这个项目的 URLs。对于一个存在的本地版本库，这个命令将会向主机位置添加一个叫 `gitlab` 的远程仓库：

```
$ git remote add gitlab https://server/namespace/project.git
```

如果你的本地没有版本库的副本，你可以这样做：

```
$ git clone https://server/namespace/project.git
```

web 用户界面提供了几个有用的获取版本库信息的网页。每一个项目的主页都显示了最近的活动，并且通过顶部的链接可以使你浏览项目文件以及提交日志。

一起工作

在一个 GitLab 项目上一起工作的最简单方法就是赋予协作者对 git 版本库的直接 push 权限。你可以通过项目设定的 “Members（成员）” 部分向一个项目添加写作者，并且将这个新的协作者与一个访问级别关联（不同的访问级别在 “组” 中已简单讨论）。通过赋予一个协作者 “Developer（开发者）” 或者更高的访问级别，这个用户就可以毫无约束地直接向版本库或者向分支进行提交。

另外一个让合作更解耦的方法就是使用合并请求。它的优点在于让任何能够看到这个项目的协作者在被管控的情况下对这个项目作出贡献。可以直接访问的协作者能够简单的创建一个分支，向这个分支进行提交，也可以开启一个向 `master` 或者其他任何一个分支的合并请求。对版本库没有推送权限的协作者则可以 “fork” 这个版本库（即创建属于自己的这个库的副本），向 那个 副本进行提交，然后从那个副本开启一个到主项目的合并请求。这个模型使得项目拥有者完全控制着向版本库的提交，以及什么时候允许加入陌生协作者的贡献。

在 GitLab 中合并请求和问题是一个长久讨论的主要部分。每一个合并请求都允许在提出改变的行进行讨论（它支持一个轻量级的代码审查），也允许对一个总体性话题进行讨论。两者都可以被分配给用户，或者组织到 milestones（里程碑）界面。

这个部分主要聚焦于在 GitLab 中与 Git 相关的特性，但是 GitLab 作为一个成熟的系统，它提供了许多其他产品来帮助你协同工作，例如项目 wiki 与系统维护工具。GitLab 的一个优点在于，服务器设置和运行以后，你将很少需要调整配置文件或通过 SSH 连接服务器；绝大多数的管理和日常使用都可以在浏览器界面中完成。

4.9 第三方托管的选择

第三方托管的选择

如果不想设立自己的 Git 服务器，你可以选择将你的 Git 项目托管到一个外部专业的托管网站。这带来了一些好处：一个托管网站可以用来快速建立并开始项目，且无需进行服务器维护和监控工作。即使你在内部设立并且运行了自己的服务器，你仍然可以把你的开源代码托管在公共托管网站 - 这通常更有助于开源社区来发现和帮助你。

现在，有非常多的托管供你选择，每个选择都有不同的优缺点。欲查看最新列表，请浏览 Git 维基的 GitHosting 页面 https://git.wiki.kernel.org/index.php/GitHosting

我们会在 Chapter 6 详细讲解 GitHub，作为目前最大的 Git 托管平台，你很可能需要与托管在 GitHub 上的项目进行交互，而且你也很可能并不想去设立你自己的 Git 服务器。

4.10 总结

总结

你有多种远程存取 Git 仓库的选择便于与其他人合作或是分享你的工作。

运行你自己的服务器将有许多权限且允许你运行该服务于你自己的防火墙内，但如此通常需要耗费你大量的时间去设置与维护服务器。如果你放置你的资料于托管服务器内，可轻易的设置与维护；无论如何，你必须能够保存你的代码在其他服务器，且某些组织不允许此作法。这将直接了当的决定哪个作法或组合的方式较适合你或你的组织。

5. 分布式 Git

第5章. 分布式 Git

你现在拥有了一个远程 Git 版本库，能为所有开发者共享代码提供服务，在一个本地工作流程下，你也已经熟悉了基本 Git 命令。你现在可以学习如何利用 Git 提供的一些分布式工作流程了。

这一章中，你将会学习如何作为贡献者或整合者，在一个分布式协作的环境中使用 Git。你会学习为一个项目成功地贡献代码，并接触一些最佳实践方式，让你和项目的维护者能轻松地完成这个过程。另外，你也会学到如何管理有很多开发者提交贡献的项目。

5.1 分布式工作流程

分布式工作流程

同传统的集中式版本控制系统（CVCS）不同，Git 的分布式特性使得开发者间的协作变得更加灵活多样。在集中式系统中，每个开发者就像是连接在集线器上的节点，彼此的工作方式大体相像。而在 Git 中，每个开发者同时扮演着节点和集线器的角色——也就是说，每个开发者既可以将自己的代码贡献到其他的仓库中，同时也能维护自己的公开仓库，让其他人可以在其基础上工作并贡献代码。由此，Git 的分布式协作可以为你的项目和团队衍生出种种不同的工作流程，接下来的章节会介绍几种利用了 Git 的这种灵活性的常见应用方式。我们将讨论每种方式的优点以及可能的缺点；你可以选择使用其中的某一种，或者将它们的特性混合搭配使用。

集中式工作流

集中式系统中通常使用的是单点协作模型——集中式工作流。一个中心集线器，或者说仓库，可以接受代码，所有人将自己的工作与之同步。若干个开发者则作为节点——也就是中心仓库的消费者——并且与其进行同步。

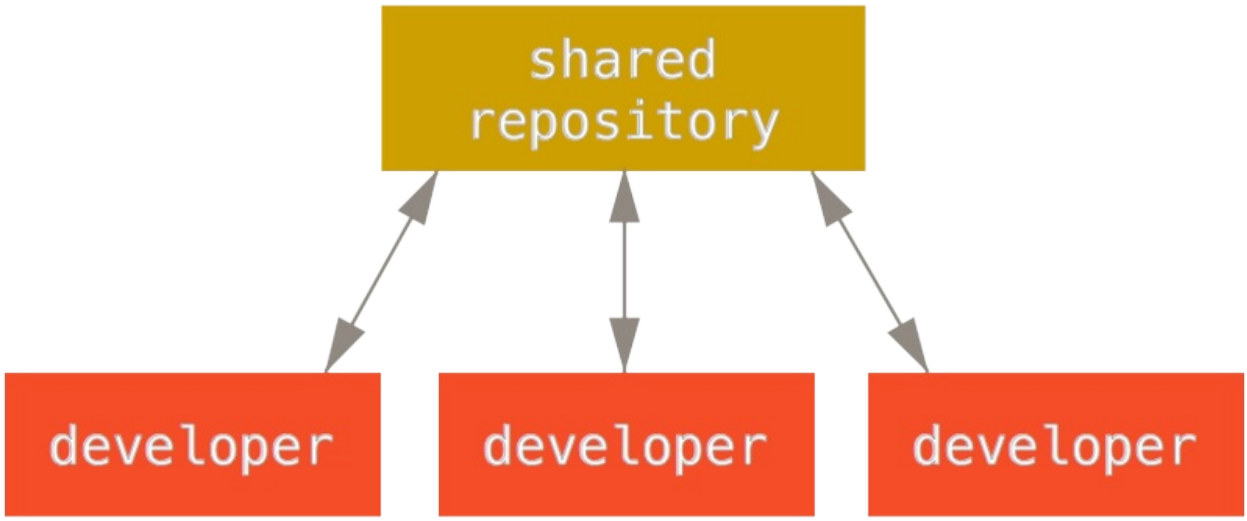


Figure 5-1. 集中式工作流。

这意味着如果两个开发者从中心仓库克隆代码下来，同时作了一些修改，那么只有第一个开发者可以顺利地把数据推送回共享服务器。第二个开发者在推送修改之前，必须先将第一个人的工作合并进来，这样才不会覆盖第一个人的修改。这和 Subversion （或任何 CVCS）中的概念一样，而且这个模式也可以很好地运用到 Git 中。

如果在公司或者团队中，你已经习惯了使用这种集中式工作流程，完全可以继续采用这种简单的模式。只需要搭建好一个中心仓库，并给开发团队中的每个人推送数据的权限，就可以开展工作了。Git 不会让用

户覆盖彼此的修改。例如 John 和 Jessica 同时开始工作。John 完成了他的修改并推送到服务器。接着 Jessica 尝试提交她自己的修改，却遭到服务器拒绝。她被告知她的修改正通过非快进式（non-fast-forward）的方式推送，只有将数据抓取下来并且合并后方能推送。这种模式的工作流程的使用非常广泛，因为大多数人对其很熟悉也很习惯。

当然这并不局限于小团队。利用 Git 的分支模型，通过同时在多个分支上工作的方式，即使是上百人的开发团队也可以很好地在单个项目上协作。

集成管理者工作流

Git 允许多个远程仓库存在，使得这样一种工作流成为可能：每个开发者拥有自己仓库的写权限和其他所有人仓库的读权限。这种情形下通常会有个代表“官方”项目的权威的仓库。要为这个项目做贡献，你需要从该项目克隆出一个自己的公开仓库，然后将自己的修改推送上去。接着你可以请求官方仓库的维护者拉取更新合并到主项目。维护者可以将你的仓库作为远程仓库添加进来，在本地测试你的变更，将其合并入他们的分支并推送回官方仓库。这一流程的工作方式如下所示（见 Figure 5-2）：

1. 项目维护者推送到主仓库。
2. 贡献者克隆此仓库，做出修改。
3. 贡献者将数据推送到自己的公开仓库。
4. 贡献者给维护者发送邮件，请求拉取自己的更新。
5. 维护者在自己本地的仓库中，将贡献者的仓库加为远程仓库并合并修改。
6. 维护者将合并后的修改推送到主仓库。

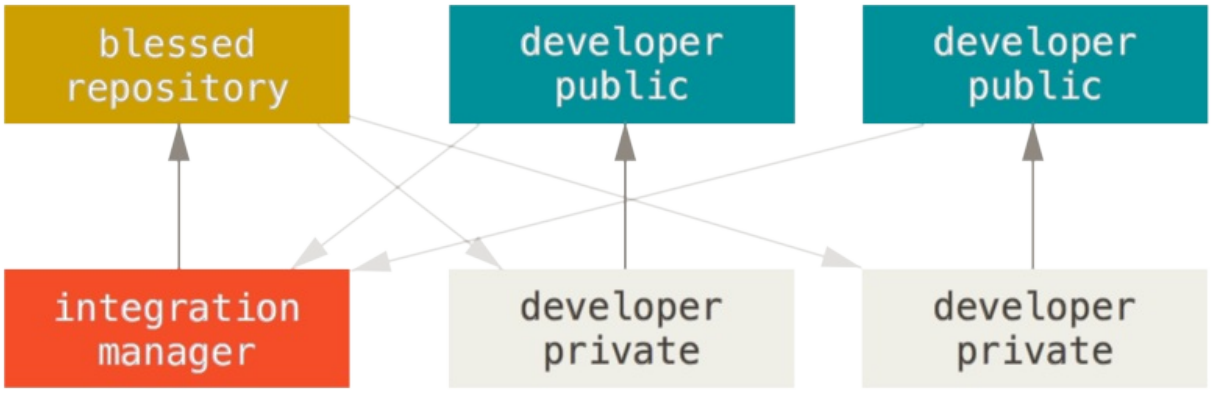


Figure 5-2. 集成管理者工作流。

这是 GitHub 和 GitLab 等集线器式（hub-based）工具最常用的工作流程。人们可以容易地将某个项目派生成为自己的公开仓库，向这个仓库推送自己的修改，并为每个人所见。这么做最主要的优点之一是你可以持续地工作，而主仓库的维护者可以随时拉取你的修改。贡献者不必等待维护者处理完提交的更新——每一方都可以按照自己节奏工作。

司令官与副官工作流

这其实是多仓库工作流程的变种。一般拥有数百位协作开发者的超大型项目才会用到这样的工作方式，例如著名的 Linux 内核项目。被称为副官（lieutenant）的各个集成管理者分别负责集成项目中的特定部分。所有这些副官头上还有一位称为司令官（dictator）的总集成管理者负责统筹。司令官维护的仓库作为参考仓库，为所有协作者提供他们需要拉取的项目代码。整个流程看起来是这样的(见 Figure 5-3):

1. 普通开发者在自己的特性分支上工作，并根据 `master` 分支进行变基。这里是司令官的 `master` 分支。
2. 副官将普通开发者的特性分支合并到自己的 `master` 分支中。
3. 司令官将所有副官的 `master` 分支并入自己的 `master` 分支中。
4. 司令官将集成后的 `master` 分支推送到参考仓库中，以便所有其他开发者以此为基础进行变基。

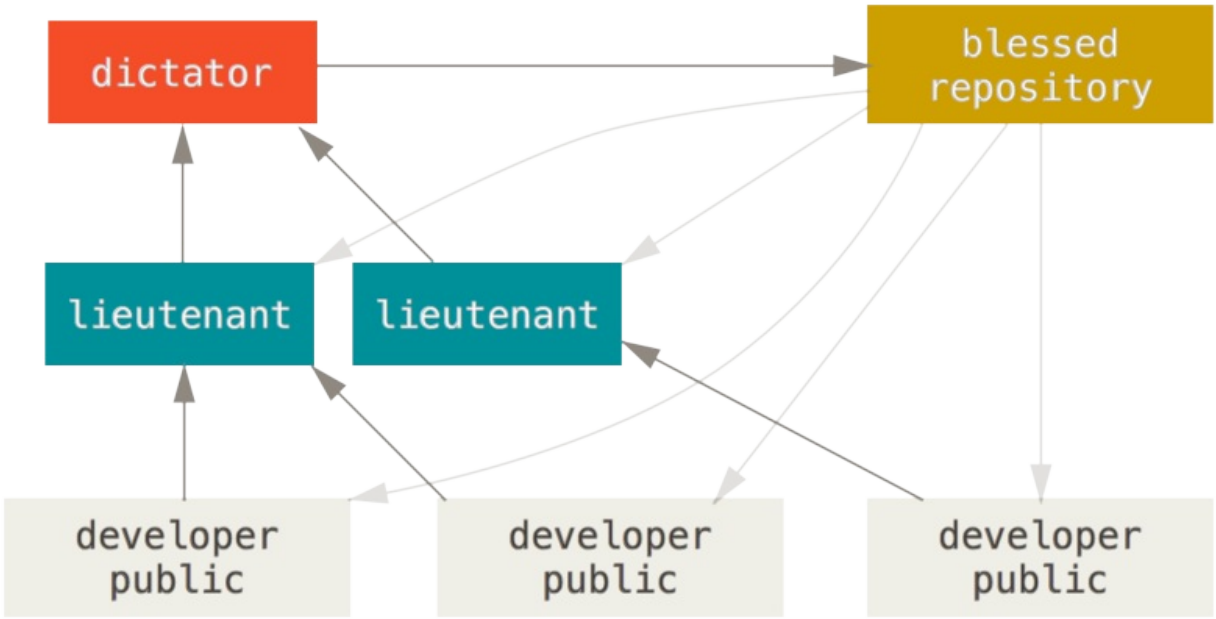


Figure 5-3. 司令官与副官工作流。

这种工作流程并不常用，只有当项目极为庞杂，或者需要多级别管理时，才会体现出优势。利用这种方式，项目总负责人（即司令官）可以把大量分散的集成工作委托给不同的小组负责人分别处理，然后在不同时刻将大块的代码子集统筹起来，用于之后的整合。

工作流程总结

上面介绍了在 Git 等分布式系统中经常使用的工作流程，但是在实际的开发中，你会遇到许多可能适合你的特定工作流程的变种。现在你应该已经清楚哪种工作流程组合可能比较适合你了，我们会给出一些如何扮演不同工作流程中主要角色的更具体的例子。下一节我们将会学习为项目做贡献的一些常用模式。

5.2 向一个项目贡献

向一个项目贡献

描述如何向一个项目贡献的主要困难在于完成贡献有很多不同的方式。因为 Git 非常灵活，人们可以通过不同的方式来一起工作，所以描述应该如何贡献并不是非常准确 - 每一个项目都有一点儿不同。影响因素包括活跃贡献者的数量、选择的工作流程、提交权限与可能包含的外部贡献方法。

第一个影响因素是活跃贡献者的数量 - 积极地向这个项目贡献代码的用户数量以及他们的贡献频率。在许多情况下，你可能会有两三个开发者一天提交几次，对于不活跃的项目可能更少。对于大一些的公司或项目，开发者的数量可能会是上千，每天都有成百上千次提交。这很重要，因为随着开发者越来越多，在确保你的代码能干净地应用或轻松地合并时会遇到更多问题。提交的改动可能表现为过时的，也可能在你正在做改动或者等待改动被批准应用时被合并入的工作严重损坏。如何保证代码始终是最新的，并且提交始终是有效的？

下一个影响因素是项目使用的工作流程。它是中心化的吗，即每一个开发者都对主线代码有相同的写入权限？项目是否有一个检查所有补丁的维护者或整合者？是否所有的补丁是同行评审后批准的？你是否参与了那个过程？是否存在副官系统，你必须先将你的工作提交到上面？

下一个问题是提交权限。是否有项目的写权限会使向项目贡献所需的流程有极大的不同。如果没有写权限，项目会选择何种方式接受贡献的工作？是否甚至有一个如何贡献的规范？你一次贡献多少工作？你多久贡献一次？

所有这些问题都会影响实际如何向一个项目贡献，以及对你来说哪些工作流程更适合或者可用。我们将会由浅入深，通过一系列用例来讲述其中的每一个方面；从这些例子应该能够建立实际中你需要的特定工作流程。

提交准则

在我们开始查看特定的用例前，这里有一个关于提交信息的快速说明。有一个好的创建提交的准则并且坚持使用会让与 Git 工作和与其他人协作更容易。Git 项目提供了一个文档，其中列举了关于创建提交到提交补丁的若干好的提示 - 可以在 Git 源代码中的 `Documentation/SubmittingPatches` 文件中阅读它。

首先，你不会想要把空白错误（根据 git help diff 的描述，结合下面给出的图片，空白错误是指行尾的空格、Tab 制表符，和行首空格后跟 Tab 制表符的行为）提交上去。Git 提供了一个简单的方式来检查这点 - 在提交前，运行 `git diff --check` ，它将会找到可能的空白错误并将它们为你列出来。

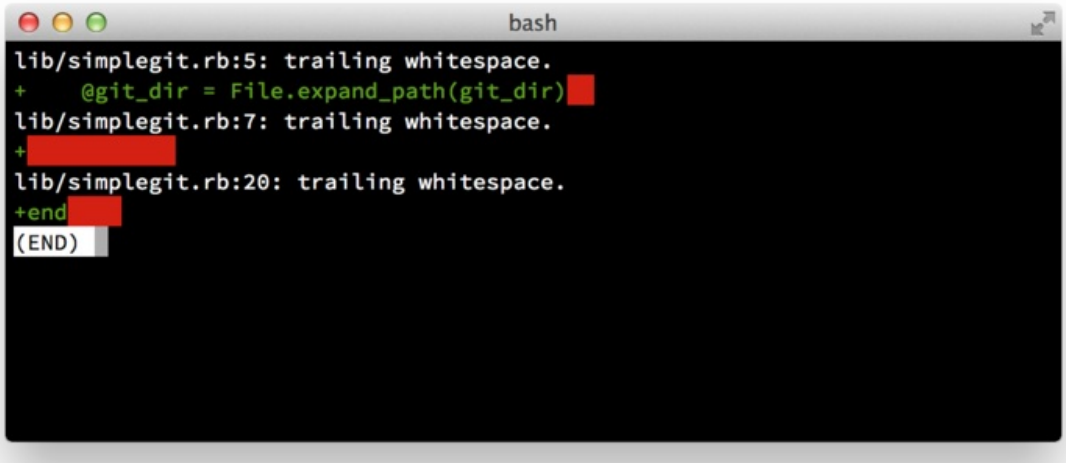


Figure 5-4. `git diff --check` 的输出

如果在提交前运行那个命令，可以知道提交中是否包含可能会使其他开发者恼怒的空白问题。

接下来，尝试让每一个提交成为一个逻辑上的独立变更集。如果可以，尝试让改动可以理解 - 不要在整个周末编码解决五个问题，然后在周一时将它们提交为一个巨大的提交。即使在周末期间你无法提交，在周一时使用暂存区域将你的工作最少拆分为每个问题一个提交，并且为每一个提交附带一个有用的信息。如果其中一些改动修改了同一个文件，尝试使用 `git add --patch` 来部分暂存文件（在 “交互式暂存” 中有详细介绍）。不管你做一个或五个提交，只要所有的改动是在同一时刻添加的，项目分支末端的快照就是独立的，使同事开发者必须审查你的改动时尽量让事情容易些。当你之后需要时这个方法也会使拉出或还原一个变更集更容易些。 “重写历史” 描述了重写历史与交互式暂存文件的若干有用的 Git 技巧 - 在将工作发送给其他人前使用这些工具来帮助生成一个干净又易懂的历史。

最后一件要牢记的事是提交信息。有一个创建优质提交信息的习惯会使 Git 的使用与协作容易的多。一般情况下，信息应当以少于 50 个字符（25个汉字）的单行开始且简要地描述变更，接着是一个空白行，再接着是一个更详细的解释。Git 项目要求一个更详细的解释，包括做改动的动机和它的实现与之前行为的对比 - 这是一个值得遵循的好规则。在这些信息中使用现在时态祈使语气也是一个好想法。换句话说，使用命令。使用 “Add tests for.” 而不是 “I added tests for” 或 “Adding tests for,” 。这里是一份最初由 Tim Pope 写的模板：

```
修改的摘要（50 个字符或更少）

如果必要的话，加入更详细的解释文字。在
大概 72 个字符的时候换行。在某些情形下，
第一行被当作一封电子邮件的标题，剩下的
文本作为正文。分隔摘要与正文的空行是
必须的（除非你完全省略正文）；如果你将
两者混在一起，那么类似变基等工具无法
```

```
正常工作。

空行接着更进一步的段落。

  - 句号也是可以的。

  - 项目符号可以使用典型的连字符或星号
    前面一个空格，之间用空行隔开，
    但是可以依据不同的惯例有所不同。
```

如果你所有的提交信息看起来都像这样，对你与跟你工作在一起的其他开发者来说事情会变得非常容易。Git 项目有一个良好格式化的提交信息 - 尝试在那儿运行 `git log --no-merges` 来看看漂亮的格式化的项目提交历史像什么样。

在接下来的例子中，以及贯穿本书大部分，出于简洁性的原因本书不会有像这样漂亮格式的信息；相反，我们使用 `-m` 选项的 `git commit`。照我们说的做，而不是照我们做的做。

私有小型团队

你可能会遇到的最简单的配置是有一两个其他开发者的私有项目。“私有” 在这个上下文中，意味着闭源 - 不可以从外面的世界中访问到。你和其他的开发者都有仓库的推送权限。

在这个环境下，可以采用一个类似使用 Subversion 或其他集中式的系统时会使用的工作流程。依然可以得到像离线提交、非常容易地新建分支与合并分支等高级功能，但是工作流程可以是很简单的；主要的区别是合并发生在客户端这边而不是在提交时发生在服务器那边。让我们看看当两个开发者在一个共享仓库中一起工作时会是什么样子。第一个开发者，John，克隆了仓库，做了改动，然后本地提交。（为了缩短这些例子长度，协议信息已被替换为 `...`。）

```
# John's Machine
$ git clone john@githost:simplegit.git
Initialized empty Git repository in /home/john/simplegit/.git/
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'removed invalid default value'
[master 738ee87] removed invalid default value
 1 files changed, 1 insertions(+), 1 deletions(-)
```

第二个开发者，Jessica，做了同样的事情 - 克隆仓库并提交了一个改动：

```
# Jessica's Machine
$ git clone jessica@githost:simplegit.git
Initialized empty Git repository in /home/jessica/simplegit/.git/
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'add reset task'
[master fbff5bc] add reset task
 1 files changed, 1 insertions(+), 0 deletions(-)
```

现在，Jessica 把她的工作推送到服务器上：

```
# Jessica's Machine
$ git push origin master
...
To jessica@githost:simplegit.git
   1edee6b..fbff5bc  master -> master
```

John 也尝试推送他的改动：

```
# John's Machine
$ git push origin master
To john@githost:simplegit.git
 ! [rejected]        master -> master (non-fast forward)
error: failed to push some refs to 'john@githost:simplegit.git'
```

不允许 John 推送是因为在同一时间 Jessica 已经推送了。如果之前习惯于用 Subversion 那么理解这点特别重要，因为你会注意到两个开发者并没有编辑同一个文件。尽管 Subversion 会对编辑的不同文件在服务器上自动进行一次合并，但 Git 要求你在本地合并提交。John 必须抓取 Jessica 的改动并合并它们，才能被允许推送。

```
$ git fetch origin
...
From john@githost:simplegit
 + 049d078...fbff5bc master     -> origin/master
```

在这个时候，John 的本地仓库看起来像这样：

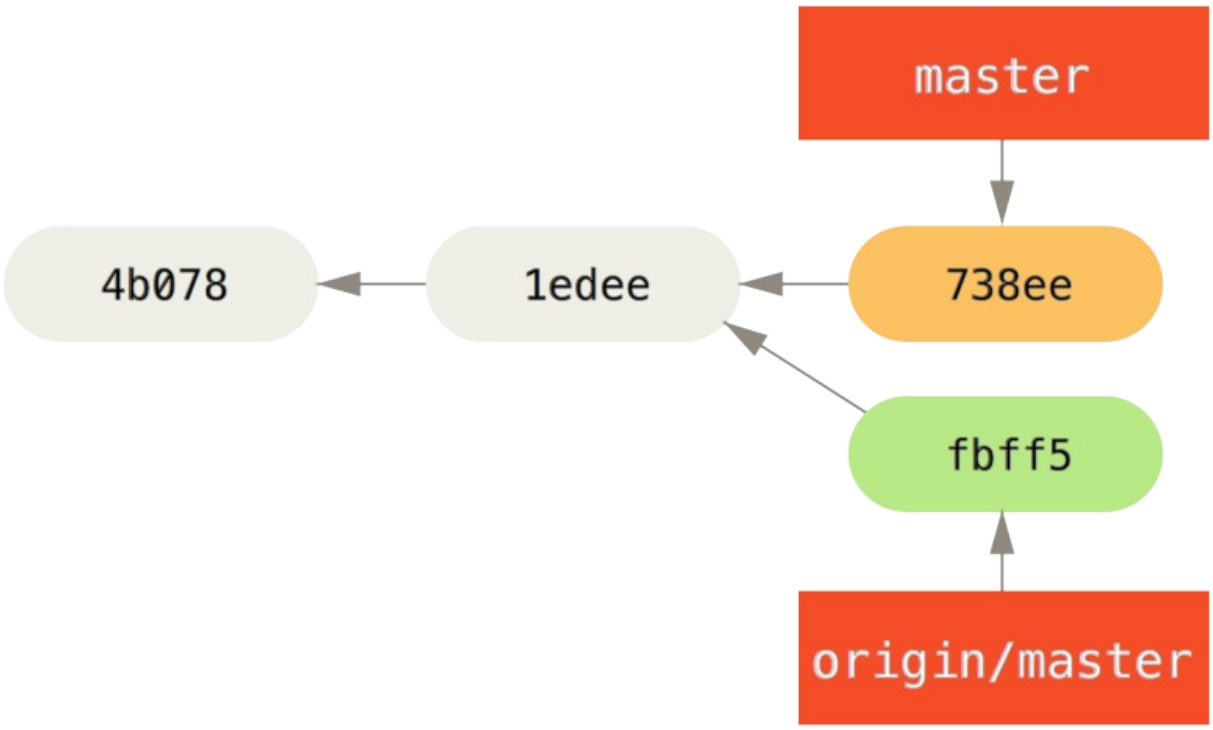


Figure 5-5. John 的分叉历史

John 有一个引用指向 Jessica 推送上去的改动，但是他必须将它们合并入自己的工作中之后才能被允许推送。

```
$ git merge origin/master
Merge made by recursive.
 TODO |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

合并进行地很顺利 - John 的提交历史现在看起来像这样：

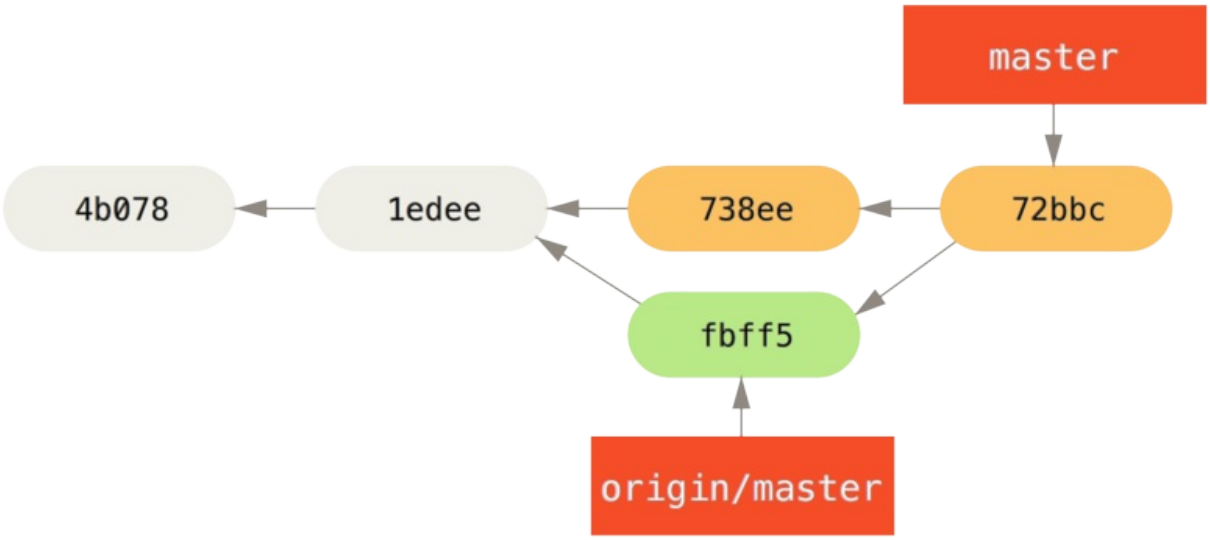


Figure 5-6. 合并了 `origin/master` 之后 John 的仓库

现在，John 可以测试代码，确保它依然正常工作，然后他可以把合并的新工作推送到服务器上：

```
$ git push origin master
...
To john@githost:simplegit.git
   fbff5bc..72bbc59  master -> master
```

最终，John 的提交历史看起来像这样：

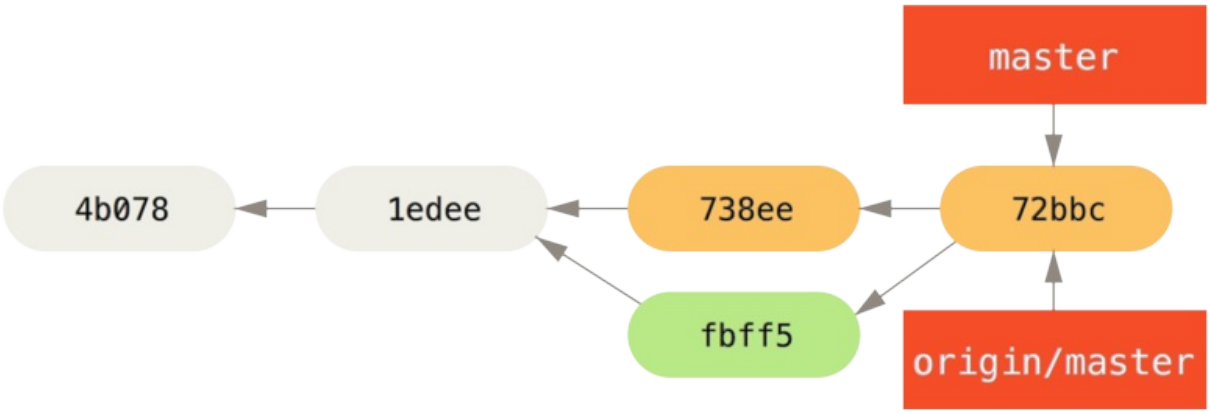


Figure 5-7. 推送到 `origin` 服务器后 John 的历史

在此期间，Jessica 在一个特性分支上工作。她创建了一个称作 `issue54` 的特性分支并且在那个分支上做了三次提交。她还没有抓取 John 的改动，所以她的提交历史看起来像这样：

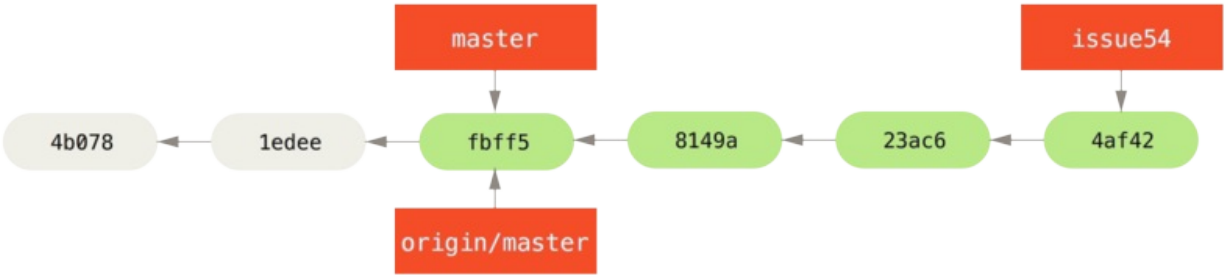


Figure 5-8. Jessica 的特性分支

Jessica 想要与 John 同步，所以她进行了抓取操作：

```
# Jessica's Machine
$ git fetch origin
...
From jessica@githost:simplegit
   fbff5bc..72bbc59  master     -> origin/master
```

那会同时拉取 John 推送的工作。Jessica 的历史现在看起来像这样：

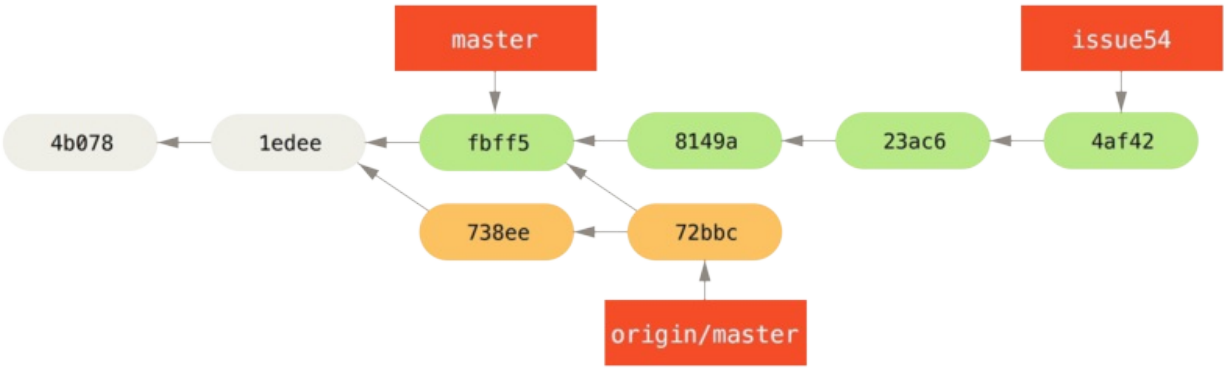


Figure 5-9. 抓取 John 的改动后 Jessica 的历史

Jessica 认为她的特性分支已经准备好了，但是她想要知道必须合并什么进入她的工作才能推送。她运行 `git log` 来找出：

```
$ git log --no-merges issue54..origin/master
commit 738ee872852dfaa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

   removed invalid default value
```

`issue54..origin/master` 语法是一个日志过滤器，要求 Git 只显示所有在后面分支（在本例中是 `origin/master`）但不在前面分支（在本例中是 `issue54`）的提交的列表。我们将会在 “提交区

间” 中详细介绍这个语法。

目前，我们可以从输出中看到有一个 John 生成的但是 Jessica 还没有合并入的提交。如果她合并 `origin/master` ，也就是说将会修改她的本地工作的那个单个提交。

现在，Jessica 可以合并她的特性工作到她的 master 分支，合并 John 的工作（ `origin/master` ）进入她的 `master` 分支，然后再次推送回服务器。首先，为了整合所有这些工作她切换回她的 master 分支。

```
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

她既可以先合并 `origin/master` 也可以先合并 `issue54` - 它们都是上游，所以顺序并没有关系。不论她选择的顺序是什么最终的结果快照是完全一样的；只是历史会有一点轻微的区别。她选择先合并入 `issue54` ：

```
$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README           |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

没有发生问题；如你所见它是一次简单的快进。现在 Jessica 合并入 John 的工作（ `origin/master` ）：

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

每一个文件都干净地合并了，Jessica 的历史看起来像这样：

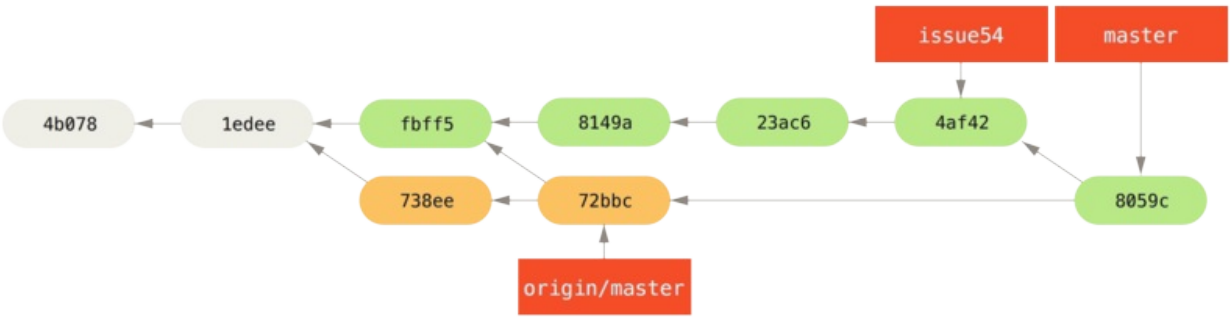


Figure 5-10. 合并了 John 的改动后 Jessica 的历史

现在 `origin/master` 是可以从 Jessica 的 `master` 分支到达的，所以她应该可以成功地推送（假设同一时间 John 并没有再次推送）：

```
$ git push origin master
...
To jessica@githost:simplegit.git
   72bbc59..8059c15  master -> master
```

每一个开发者都提交了几次并成功地合并了其他人的工作。

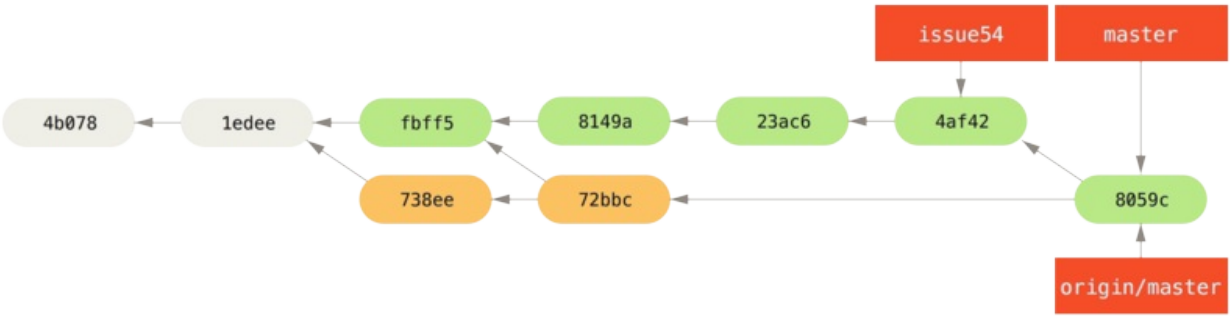


Figure 5-11. 推送所有的改动回服务器后 Jessica 的历史

这是一个最简单的工作流程。你通常在一个特性分支工作一会儿，当它准备好整合时合并回你的 master 分支。当想要共享工作时，将其合并回你自己的 master 分支，如果有改动的话然后抓取并合并 `origin/master`，最终推送到服务器上的 `master` 分支。通常顺序像这样：

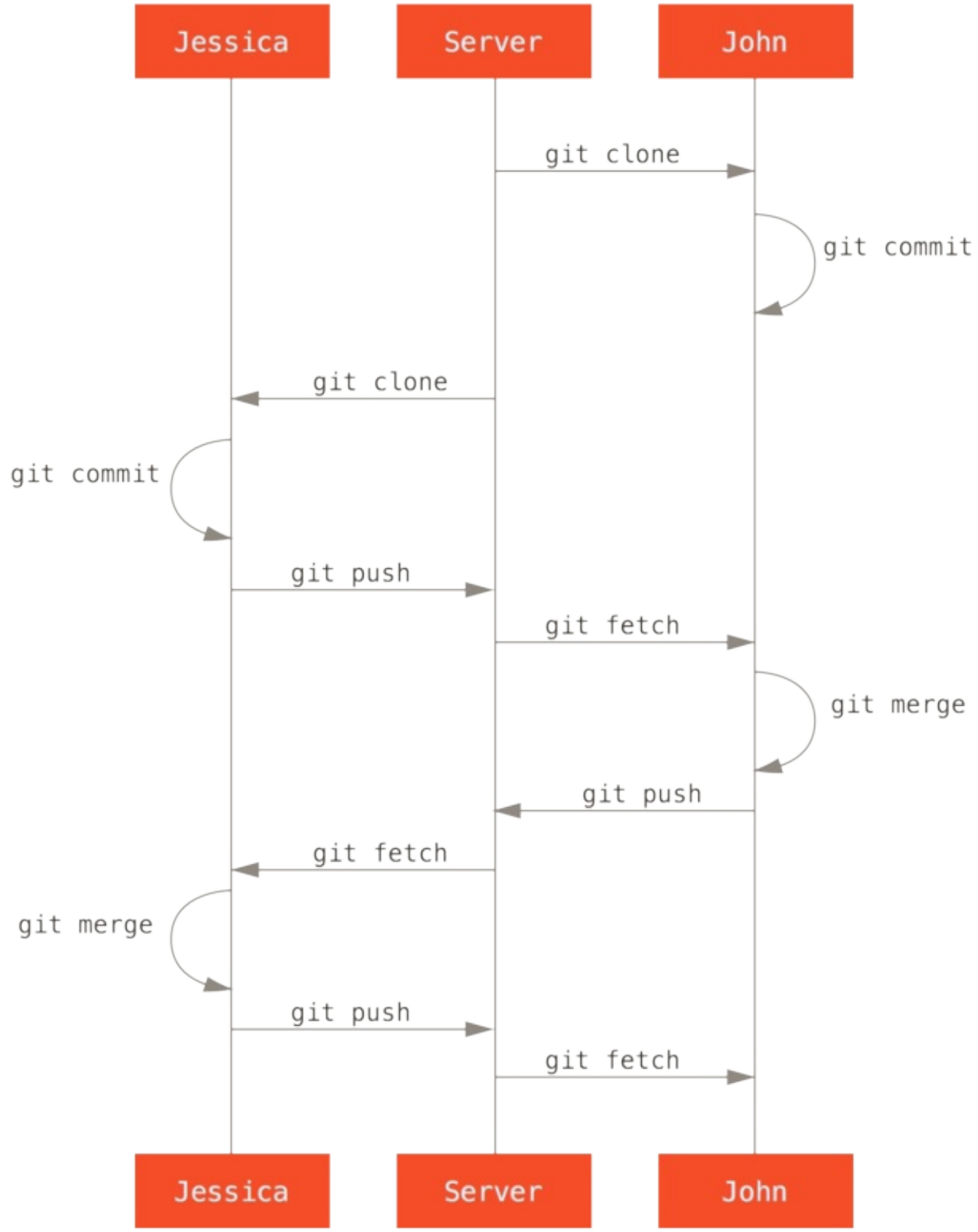


Figure 5-12. 一个简单的多人 Git 工作流程的通常事件顺序

私有管理团队

在接下来的情形中，你会看到大型私有团队中贡献者的角色。在你将学习到的这种工作环境中，小组基于

特性进行协作，这些团队的贡献将会由其他人整合。

让我们假设 John 与 Jessica 在一个特性上工作，同时 Jessica 与 Josie 在第二个特性上工作。在本例中，公司使用了一种整合-管理者工作流程，独立小组的工作只能被特定的工程师整合，主仓库的 `master` 分支只能被那些工程师更新。在这种情况下，所有的工作都是在基于团队的分支上完成的并且稍后会被整合者拉到一起。

因为 Jessica 在两个特性上工作，并且平行地与两个不同的开发者协作，让我们跟随她的工作流程。假设她已经克隆了仓库，首先决定在 `featureA` 上工作。她为那个特性创建了一个新分支然后在那做了一些工作：

```
# Jessica's Machine
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ vim lib/simplegit.rb
$ git commit -am 'add limit to log function'
[featureA 3300904] add limit to log function
 1 files changed, 1 insertions(+), 1 deletions(-)
```

在这个时候，她需要将工作共享给 John，所以她推送了 `featureA` 分支的提交到服务器上。Jessica 没有 `master` 分支的推送权限 - 只有整合者有 - 所以为了与 John 协作必须推送另一个分支。

```
$ git push -u origin featureA
...
To jessica@githost:simplegit.git
 * [new branch]      featureA -> featureA
```

Jessica 向 John 发邮件告诉他已经推送了一些工作到 `featureA` 分支现在可以看一看。当她等待 John 的反馈时，Jessica 决定与 Josie 开始在 `featureB` 上工作。为了开始工作，她基于服务器的 `master` 分支开始了一个新分支。

```
# Jessica's Machine
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch 'featureB'
```

现在，Jessica 在 `featureB` 分支上创建了几次提交：

```
$ vim lib/simplegit.rb
$ git commit -am 'made the ls-tree function recursive'
[featureB e5b0fdc] made the ls-tree function recursive
 1 files changed, 1 insertions(+), 1 deletions(-)
$ vim lib/simplegit.rb
$ git commit -am 'add ls-files'
[featureB 8512791] add ls-files
 1 files changed, 5 insertions(+), 0 deletions(-)
```

Jessica 的仓库看起来像这样：

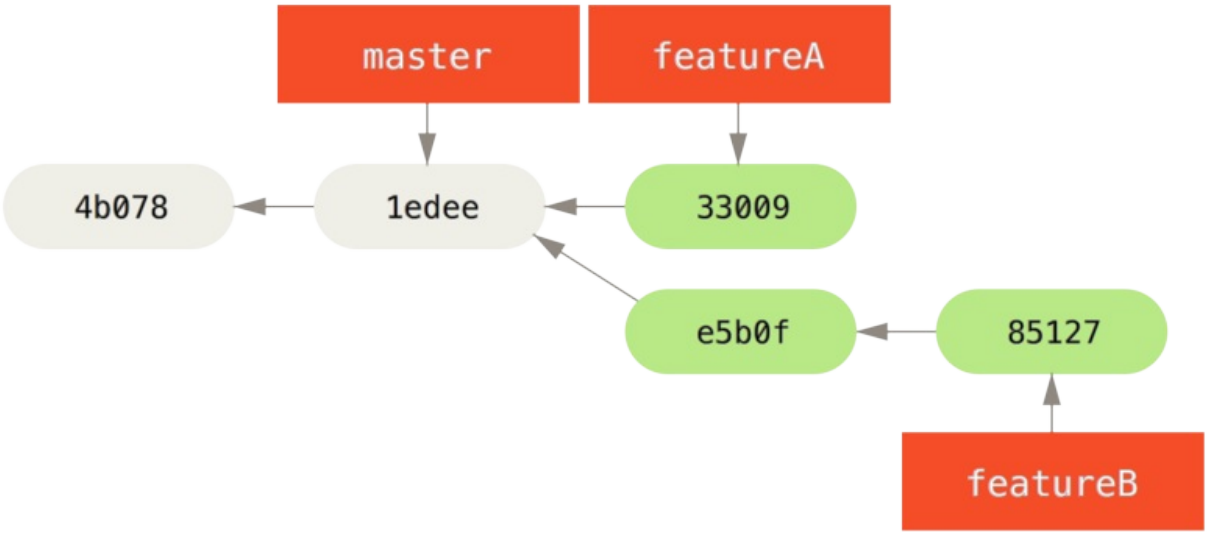


Figure 5-13. Jessica 的初始提交历史

她准备好推送工作了，但是一封来自 Josie 的邮件告知一些初始工作已经被推送到服务器上的 `featureBee` 上了。Jessica 在能推送到服务器前首先需要将那些改动与她自己的合并。然后她可以通过 `git fetch` 抓取 Josie 的改动：

```
$ git fetch origin
...
From jessica@githost:simplegit
 * [new branch]      featureBee -> origin/featureBee
```

Jessica 现在可以通过 `git merge` 将其合并到她做的工作中：

```
$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    4 ++++
 1 files changed, 4 insertions(+), 0 deletions(-)
```

有点儿问题 - 她需要将在 `featureB` 分支上合并的工作推送到服务器上的 `featureBee` 分支。她可以通过指定本地分支加上冒号（:）加上远程分支给 `git push` 命令来这样做：

```
$ git push -u origin featureB:featureBee
...
To jessica@githost:simplegit.git
   fba9af8..cd685d1  featureB -> featureBee
```

这称作一个 引用规格。查看 “引用规格” 了解关于 Git 引用规格与通过它们可以做的不同的事情的详细

讨论。也要注意 `-u` 标记；这是 `--set-upstream` 的简写，该标记会为之后轻松地推送与拉取配置分支。

紧接着，John 发邮件给 Jessica 说他已经推送了一些改动到 `featureA` 分支并要求她去验证它们。她运行一个 `git fetch` 来拉取下那些改动：

```
$ git fetch origin
...
From jessica@githost:simplegit
   3300904..aad881d  featureA   -> origin/featureA
```

然后，通过 `git log` 她可以看到哪些发生了改变：

```
$ git log featureA..origin/featureA
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 19:57:33 2009 -0700

    changed log output to 30 from 25
```

最终，她合并 John 的工作到她自己的 `featureA` 分支：

```
$ git checkout featureA
Switched to branch 'featureA'
$ git merge origin/featureA
Updating 3300904..aad881d
Fast forward
 lib/simplegit.rb |   10 +++++++++-
1 files changed, 9 insertions(+), 1 deletions(-)
```

Jessica 想要轻微调整一些东西，所以她再次提交然后将其推送回服务器：

```
$ git commit -am 'small tweak'
[featureA 774b3ed] small tweak
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push
...
To jessica@githost:simplegit.git
   3300904..774b3ed  featureA -> featureA
```

Jessica 的提交历史现在看起来像这样：

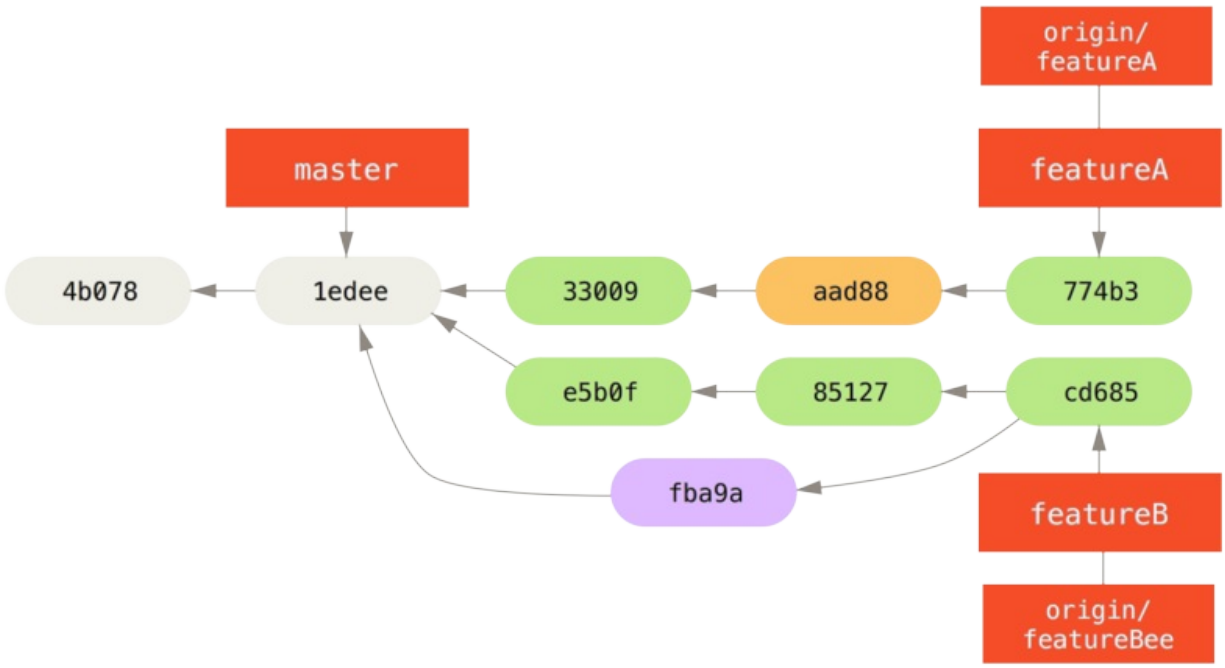


Figure 5-14. 在一个特性分支提交后 Jessica 的历史

Jessica、Josie 与 John 通知整合者在服务器上的 `featureA` 与 `featureBee` 分支准备好整合到主线中了。在整合者合并这些分支到主线后，一次抓取会拿下来一个新的合并提交，使历史看起来像这样：

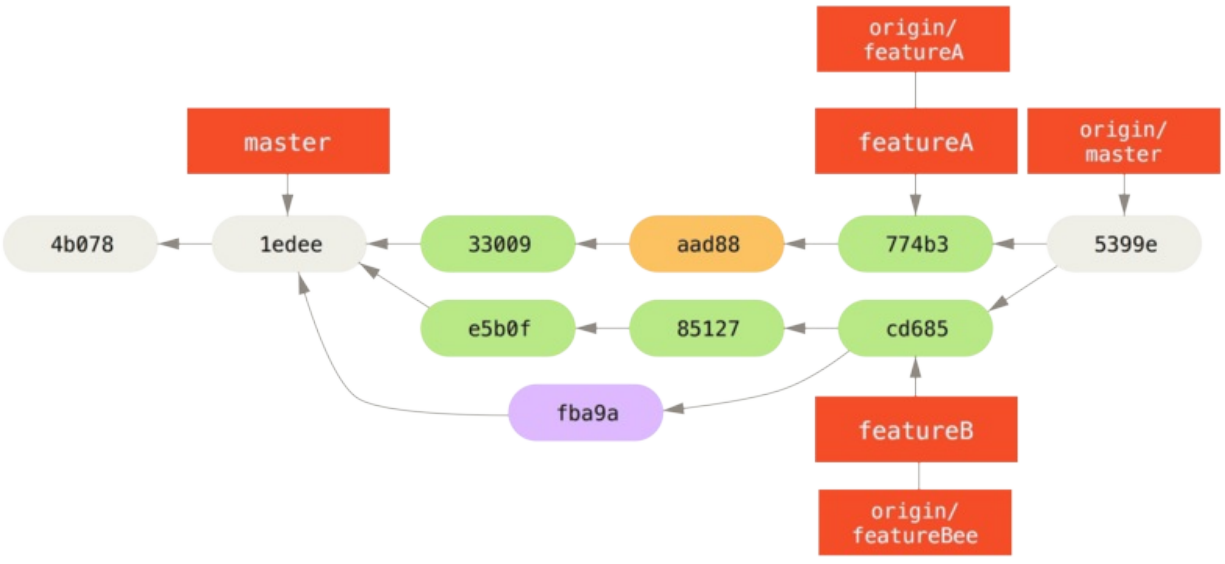


Figure 5-15. 合并了 Jessica 的两个特性分支后她的历史

许多团队切换到 Git 是因为这一允许多个团队并行工作、并在之后合并不同工作的能力。团队中更小一些的子小组可以通过远程分支协作而不必影响或妨碍整个团队的能力是 Git 的一个巨大优势。在这儿看到的

工作流程顺序类似这样：

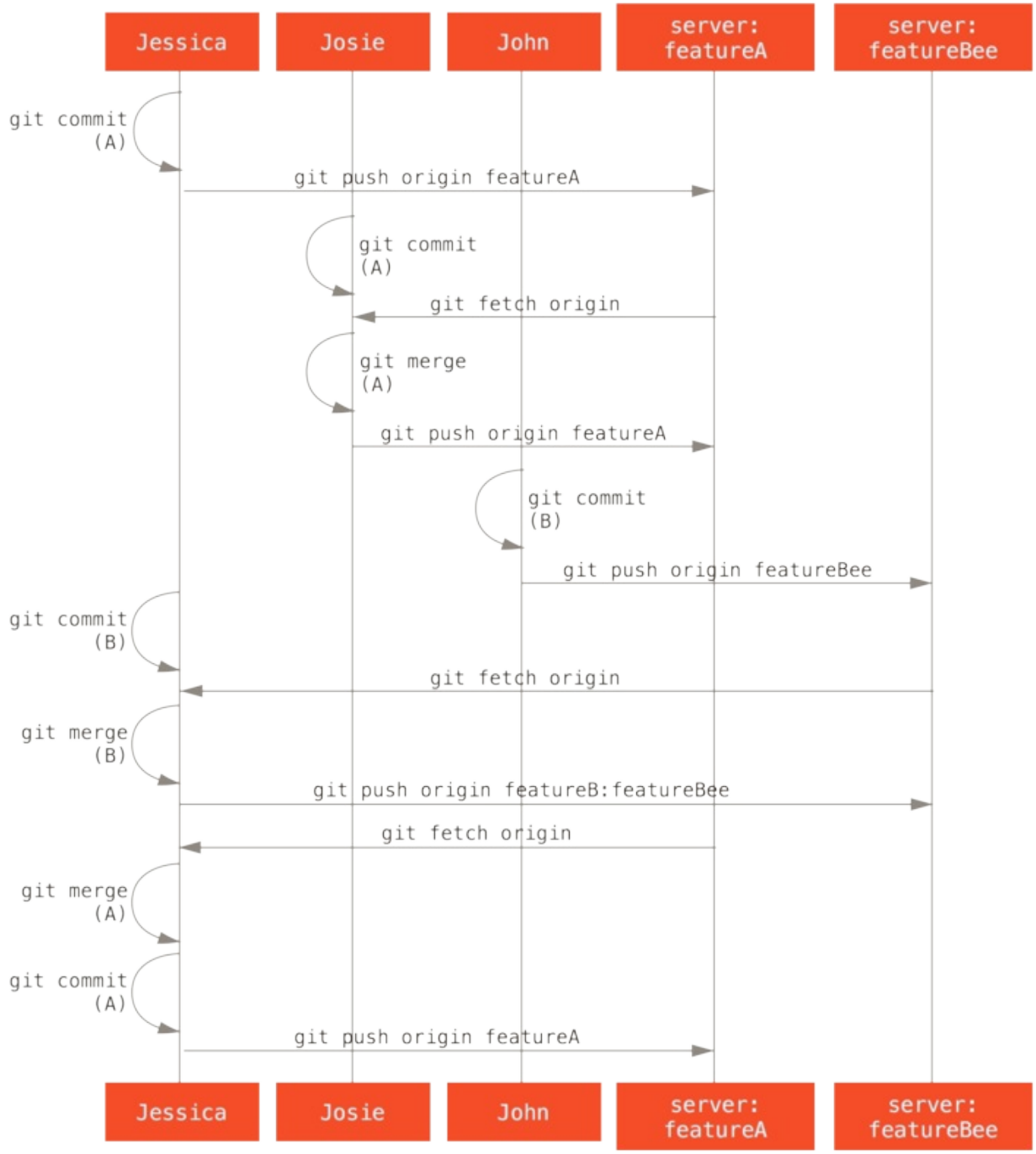


Figure 5-16. 这种管理团队工作流程的基本顺序

派生的公开项目

向公开项目做贡献有一点儿不同。因为没有权限直接更新项目的分支，你必须用其他办法将工作给维护者。第一个例子描述在支持简单派生的 Git 托管上使用派生来做贡献。许多托管站点支持这个功能（包括

GitHub、BitBucket、Google Code、repo.or.cz 等等），许多项目维护者期望这种风格的贡献。下一节会讨论偏好通过邮件接受贡献补丁的项目。

首先，你可能想要克隆主仓库，为计划贡献的补丁或补丁序列创建一个特性分支，然后在那儿做工作。顺序看起来基本像这样：

```
$ git clone (url)
$ cd project
$ git checkout -b featureA
# (work)
$ git commit
# (work)
$ git commit
```

你可能会想要使用 `rebase -i` 来将工作压缩成一个单独的提交，或者重排提交中的工作使补丁更容易被维护者审核 - 查看 “重写历史” 了解关于交互式变基的更多信息。

当你的分支工作完成后准备将其贡献回维护者，去原始项目中然后点击 “Fork” 按钮，创建一份自己的可写的项目派生仓库。然后需要添加这个新仓库 URL 为第二个远程仓库，在本例中称作 `myfork`：

```
$ git remote add myfork (url)
```

然后需要推送工作到上面。相对于合并到主分支再推送上去，推送你正在工作的特性分支到仓库上更简单。原因是工作如果不被接受或者是被拣选的，就不必回退你的 master 分支。如果维护者合并、变基或拣选你的工作，不管怎样你最终会通过拉取他们的仓库找回来你的工作。

```
$ git push -u myfork featureA
```

当工作已经被推送到你的派生后，你需要通知维护者。这通常被称作一个拉取请求（pull request），你既可以通过网站生成它 - GitHub 有它自己的 Pull Request 机制，我们将会在 Chapter 6 介绍 - 也可以运行 `git request-pull` 命令然后手动地将输出发送电子邮件给项目的维护者。

`request-pull` 命令接受特性分支拉入的基础分支，以及它们拉入的 Git 仓库 URL，输出请求拉入的所有修改的总结。例如，Jessica 想要发送给 John 一个拉取请求，她已经在刚刚推送的分支上做了两次提交。她可以运行这个：

```
$ git request-pull origin/master myfork
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
  John Smith (1):
        added a new function

are available in the git repository at:

  git://githost/simplegit.git featureA
```

```
Jessica Smith (2):
      add limit to log function
      change log output to 30 from 25

 lib/simplegit.rb |   10 +++++++++-
 1 files changed, 9 insertions(+), 1 deletions(-)
```

这个输出可以被发送给维护者 - 它告诉他们工作是从哪个分支开始、归纳的提交与从哪里拉入这些工作。

在一个你不是维护者的项目上，通常有一个总是跟踪 `origin/master` 的 `master` 分支会很方便，在特性分支上做工作是因为如果它们被拒绝时你可以轻松地丢弃。如果同一时间主仓库移动了然后你的提交不再能干净地应用，那么使工作主题独立于特性分支也会使你变基（rebase）工作时更容易。例如，你想要提供第二个特性工作到项目，不要继续在刚刚推送的特性分支上工作 - 从主仓库的 `master` 分支重新开始：

```
$ git checkout -b featureB origin/master
# (work)
$ git commit
$ git push myfork featureB
# (email maintainer)
$ git fetch origin
```

现在，每一个特性都保存在一个贮藏库中 - 类似于补丁队列 - 可以重写、变基与修改而不会让特性互相干涉或互相依赖，像这样：

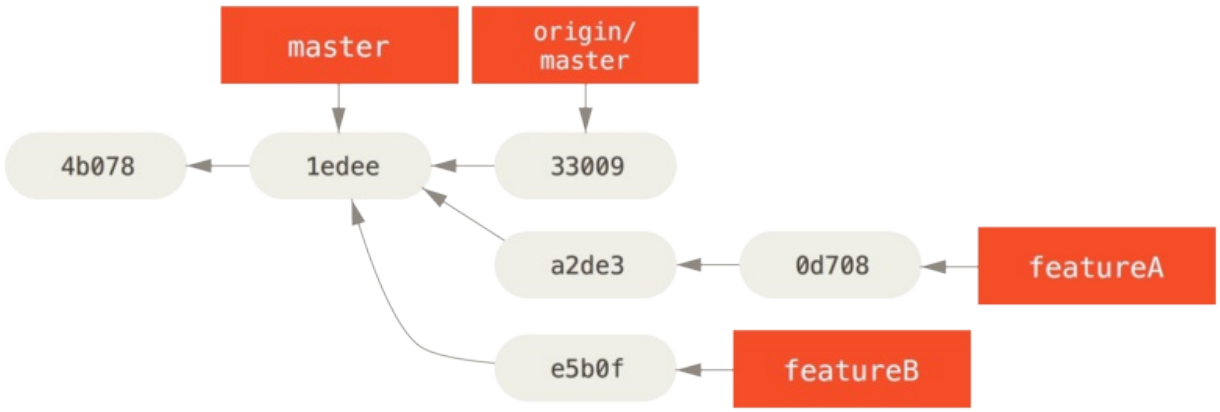


Figure 5-17. `featureB` 的初始提交历史

假设项目维护者已经拉取了一串其他补丁，然后尝试拉取你的第一个分支，但是没有干净地合并。在这种情况下，可以尝试变基那个分支到 `origin/master` 的顶部，为维护者解决冲突，然后重新提交你的改动：

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

这样会重写你的历史，现在看起来像是 Figure 5-18

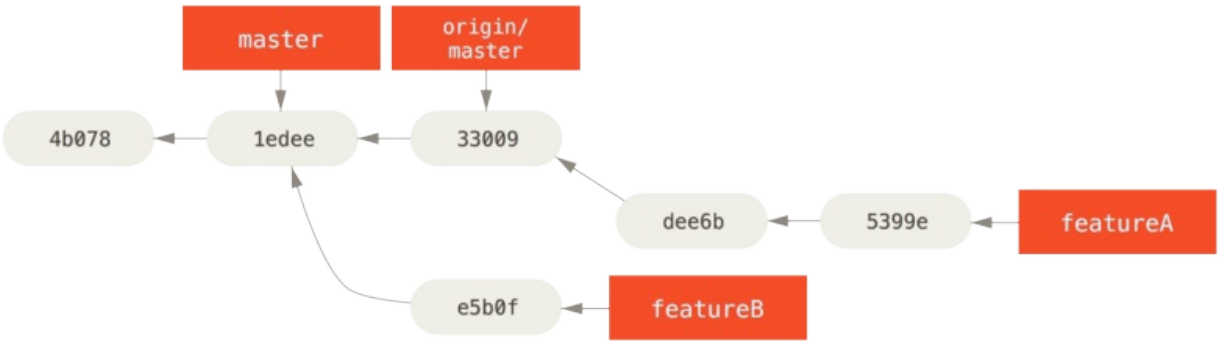


Figure 5-18. `featureA` 工作之后的提交历史

因为你将分支变基了，所以必须为推送命令指定 `-f` 选项，这样才能将服务器上有一个不是它的后代的提交的 `featureA` 分支替换掉。一个替代的选项是推送这个新工作到服务器上的一个不同分支（可能称作 `featureAv2` ）。

让我们看一个更有可能的情况：维护者看到了你的第二个分支上的工作并且很喜欢其中的概念，但是想要你修改一下实现的细节。你也可以利用这次机会将工作基于项目现在的 `master` 分支。你从现在的 `origin/master` 分支开始一个新分支，在那儿压缩 `featureB` 的改动，解决任何冲突，改变实现，然后推送它为一个新分支。

```
$ git checkout -b featureBv2 origin/master
$ git merge --no-commit --squash featureB
# (change implementation)
$ git commit
$ git push myfork featureBv2
```

`--squash` 选项接受被合并的分支上的所有工作，并将其压缩至一个变更集，使仓库变成一个真正的合并发生的状态，而不会真的生成一个合并提交。这意味着你的未来的提交将会只有一个父提交，并允许你引入另一个分支的所有改动，然后在记录一个新提交前做更多的改动。同样 `--no-commit` 选项在默认合并过程中可以用来延迟生成合并提交。

现在你可以给维护者发送一条消息，表示你已经做了要求的修改然后他们可以在你的 `featureBv2` 分支上找到那些改动。

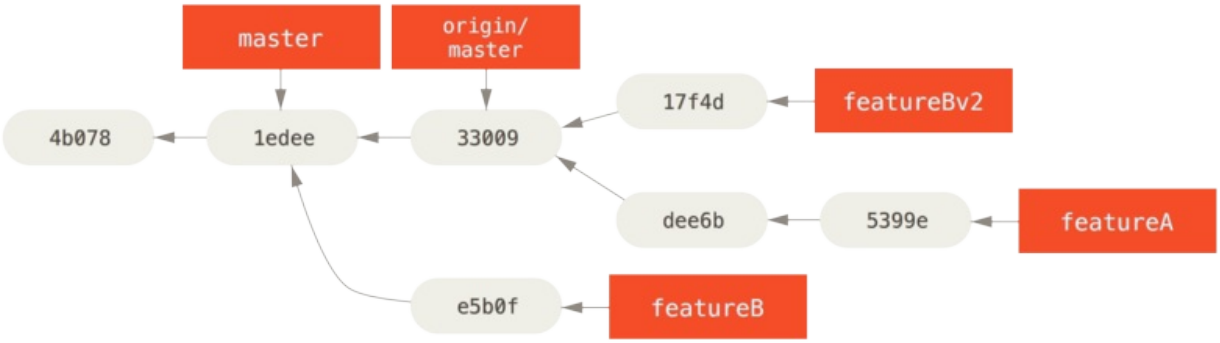


Figure 5-19. `featureBv2` 工作之后的提交历史

通过邮件的公开项目

许多项目建立了接受补丁的流程 - 需要检查每一个项目的特定规则，因为它们之间有区别。因为有几个历史悠久的、大型的项目会通过一个开发者的邮件列表接受补丁，现在我们将会通过一个例子来演示。

工作流程与之前的用例是类似的 - 你为工作的每一个补丁序列创建特性分支。区别是如何提交它们到项目中。生成每一个提交序列的电子邮件版本然后邮寄它们到开发者邮件列表，而不是派生项目然后推送到你自己的可写版本。

```
$ git checkout -b topicA
# (work)
$ git commit
# (work)
$ git commit
```

现在有两个提交要发送到邮件列表。使用 `git format-patch` 来生成可以邮寄到列表的 mbox 格式的文件 - 它将每一个提交转换为一封电子邮件，提交信息的第一行作为主题，剩余信息与提交引入的补丁作为正文。它有一个好处是是使用 `format-patch` 生成的一封电子邮件应用的提交正确地保留了所有的提交信息。

```
$ git format-patch -M origin/master
0001-add-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
```

`format-patch` 命令打印出它创建的补丁文件名字。 `-M` 开关告诉 Git 查找重命名。文件最后看起来像这样：

```
$ cat 0001-add-limit-to-log-function.patch
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20
```

```
---
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 76f47bc..f9815f1 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -14,7 +14,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log #{treeish}")
+    command("git log -n 20 #{treeish}")
   end

   def ls_tree(treeish = 'master')
--
2.1.0
```

也可以编辑这些补丁文件为邮件列表添加更多不想要在提交信息中显示出来的信息。如果在 `---` 行与补丁开头（ `diff --git` 行）之间添加文本，那么开发者就可以阅读它；但是应用补丁时会排除它。

为了将其邮寄到邮件列表，你既可以将文件粘贴进电子邮件客户端，也可以通过命令行程序发送它。粘贴文本经常会发生格式化问题，特别是那些不会合适地保留换行符与其他空白的 “更聪明的” 客户端。幸运的是，Git 提供了一个工具帮助你通过 IMAP 发送正确格式化的补丁，这可能对你更容易些。我们将会演示如何通过 Gmail 发送一个补丁，它正好是我们所知最好的邮件代理；可以在之前提到的 Git 源代码中的 `Documentation/SubmittingPatches` 文件的最下面了解一系列邮件程序的详细指令。

首先，需要在 `~/.gitconfig` 文件中设置 imap 区块。可以通过一系列的 `git config` 命令来分别设置每一个值，或者手动添加它们，不管怎样最后配置文件应该看起来像这样：

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
  user = user@gmail.com
  pass = p4ssw0rd
  port = 993
  sslverify = false
```

如果 IMAP 服务器不使用 SSL，最后两行可能没有必要，host 的值会是 `imap://` 而不是 `imaps://`。当那些设置完成后，可以使用 `git imap-send` 将补丁序列放在特定 IMAP 服务器的 Drafts 文件夹中：

```
$ cat *.patch |git imap-send
Resolving imap.gmail.com... ok
Connecting to [74.125.142.109]:993... ok
Logging in...
sending 2 messages
100% (2/2) done
```

在这个时候，你应该能够到 Drafts 文件夹中，修改收件人字段为想要发送补丁的邮件列表，可能需要抄送给维护者或负责那个部分的人，然后发送。

你也可以通过一个 SMTP 服务器发送补丁。同之前一样，你可以通过一系列的 `git config` 命令来分别设置选项，或者你可以手动地将它们添加到你的 `~/.gitconfig` 文件的 sendmail 区块：

```
[sendemail]
  smtpencryption = tls
  smtpserver = smtp.gmail.com
  smtpuser = user@gmail.com
  smtpserverport = 587
```

当这完成后，你可以使用 `git send-email` 发送你的补丁：

```
$ git send-email *.patch
0001-added-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]
Emails will be sent from: Jessica Smith <jessica@example.com>
Who should the emails be sent to? jessica@example.com
Message-ID to be used as In-Reply-To for the first email? y
```

然后，对于正在发送的每一个补丁，Git 会吐出这样的一串日志信息：

```
(mbox) Adding cc: Jessica Smith <jessica@example.com> from
  \line 'From: Jessica Smith <jessica@example.com>'
OK. Log says:
Sendmail: /usr/sbin/sendmail -i jessica@example.com
From: Jessica Smith <jessica@example.com>
To: jessica@example.com
Subject: [PATCH 1/2] added limit to log function
Date: Sat, 30 May 2009 13:29:15 -0700
Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>
X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty
In-Reply-To: <y>
References: <y>

Result: OK
```

总结

这个部分介绍了处理可能会遇到的几个迥然不同类型的 Git 项目的一些常见的工作流程，介绍了帮助管理这个过程的一些新工具。接下来，你会了解到如何在贡献的另一面工作：维护一个 Git 项目。你将会学习如何成为一个仁慈的独裁者或整合管理者。

5.3 维护项目

维护项目

除了如何有效地参与一个项目的贡献之外，你可能也需要了解如何维护项目。这包含接受并应用别人使用 `format-patch` 生成并通过电子邮件发送过来的补丁，或对项目添加的远程版本库分支中的更改进行整合。但无论是管理版本库，还是帮忙验证、审核收到的补丁，都需要同其他贡献者约定某种长期可持续的工作方式。

在特性分支中工作

如果你想向项目中整合一些新东西，最好将这些尝试局限在特性分支——一种通常用来尝试新东西的临时分支中。这样便于单独调整补丁，如果遇到无法正常工作的情况，可以先不用管，等到有时间的时候再来处理。如果你基于你所尝试进行工作的特性为分支创建一个简单的名字，比如 `ruby_client` 或者具有类似描述性的其他名字，这样即使你必须暂时抛弃它，以后回来时也不会忘记。项目的维护者一般还会为这些分支附带命名空间，比如 `sc/ruby_client`（其中 `sc` 是贡献该项工作的人名称的简写）。你应该记得，可以使用如下方式基于 master 分支建立特性分支：

```
$ git branch sc/ruby_client master
```

或者如果你同时想立刻切换到新分支上的话，可以使用 `checkout -b` 选项：

```
$ git checkout -b sc/ruby_client master
```

现在你已经准备好将别人贡献的工作加入到这个特性分支，并考虑是否将其合并到长期分支中去了。

应用来自邮件的补丁

如果你通过电子邮件收到了一个需要整合进入项目的补丁，你需要将其应用到特性分支中进行评估。有两种应用该种补丁的方法：使用 `git apply` ，或者使用 `git am` 。

使用 `apply` 命令应用补丁

如果你收到了一个使用 `git diff` 或 Unix `diff` 命令（不推荐使用这种方式，具体见下一节）创建的补丁，可以使用 `git apply` 命令来应用。假设你将补丁保存在了 `/tmp/patch-ruby-client.patch` 中，可以这样应用补丁：

```
$ git apply /tmp/patch-ruby-client.patch
```

这会修改工作目录中的文件。它与运行 `patch -p1` 命令来应用补丁几乎是等效的，但是这种方式更加严格，相对于 patch 来说，它能够接受的模糊匹配更少。它也能够处理 `git diff` 格式文件所描述的文

件添加、删除和重命名操作，而 `patch` 则不会。最后，`git apply` 命令采用了一种“全部应用，否则就全部撤销（apply all or abort all）”的模型，即补丁只有全部内容都被应用和完全不被应用两个状态，而 `patch` 可能会导致补丁文件被部分应用，最后使你的工作目录保持在一个比较奇怪的状态。总体来看，`git apply` 命令要比 `patch` 谨慎得多。并且，它不会为你创建提交——在运行之后，你需要手动暂存并提交补丁所引入的更改。

在实际应用补丁前，你还可以使用 git apply 来检查补丁是否可以顺利应用——即对补丁运行 `git apply --check` 命令：

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

如果没有产生输出，则该补丁可以顺利应用。如果检查失败了，该命令还会以一个非零的状态退出，所以需要时你也可以在脚本中使用它。

使用 `am` 命令应用补丁

如果补丁的贡献者也是一个 Git 用户，并且其能熟练使用 `format-patch` 命令来生成补丁，这样的话你的工作会变得更加轻松，因为这种补丁中包含了作者信息和提交信息供你参考。如果可能的话，请鼓励贡献者使用 `format-patch` 而不是 `diff` 来为你生成补丁。而只有对老式的补丁，你才必须使用 `git apply` 命令。

要应用一个由 `format-patch` 命令生成的补丁，你应该使用 `git am` 命令。从技术的角度看，`git am` 是为了读取 mbox 文件而构建的，mbox 是一种用来在单个文本文件中存储一个或多个电子邮件消息的简单纯文本格式。其大致格式如下所示：

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20
```

这其实就是你前面看到的 format-patch 命令输出的开始几行。而同时它也是有效的 mbox 电子邮件格式。如果有人使用 git send-email 命令将补丁以电子邮件的形式发送给你，你便可以将它下载为 mbox 格式的文件，之后将 git am 命令指向该文件，它会应用其中包含的所有补丁。如果你所使用的邮件客户端能够同时将多封邮件保存为 mbox 格式的文件，你甚至能够将一系列补丁打包为单个 mbox 文件，并利用 `git am` 命令将它们一次性全部应用。

然而，如果贡献者将 `format-patch` 生成的补丁文件上传到类似 Request Ticket 的任务处理系统，你可以先将其保存到本地，之后通过 `git am` 来应用补丁：

```
$ git am 0001-limit-log-function.patch
```

```
Applying: add limit to log function
```

你会看到补丁被顺利地应用，并且为你自动创建了一个新的提交。其中的作者信息来自于电子邮件头部的 `From` 和 `Date` 字段，提交消息则取自 `Subject` 和邮件正文中补丁之前的内容。比如，应用上面那个 mbox 示例后生成的提交是这样的：

```
$ git log --pretty=fuller -1
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Author:     Jessica Smith <jessica@example.com>
AuthorDate: Sun Apr 6 10:17:23 2008 -0700
Commit:     Scott Chacon <schacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700

   add limit to log function

   Limit log functionality to the first 20
```

其中 `Commit` 信息表示的是应用补丁的人和应用补丁的时间。 `Author` 信息则表示补丁的原作者和原本的创建时间。

但是，有时候无法顺利地应用补丁。这也许是因为你的主分支和创建补丁的分支相差较多，也有可能是因为这个补丁依赖于其他你尚未应用的补丁。这种情况下， `git am` 进程将会报错并且询问你要做什么：

```
$ git am 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

该命令将会在所有出现问题的文件内加入冲突标记，就和发生冲突的合并或变基操作一样。而你解决问题的手段很大程度上也是一样的——即手动编辑那些文件来解决冲突，暂存新的文件，之后运行 `git am --resolved` 继续应用下一个补丁：

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: seeing if this helps the gem
```

如果你希望 Git 能够尝试以更加智能的方式解决冲突，你可以对其传递 `-3` 选项来使 Git 尝试进行三方合并。该选项默认并没有打开，因为如果用于创建补丁的提交并不在你的版本库内的话，这样做是没有用处的。而如果你确实有那个提交的话——比如补丁是基于某个公共提交的——那么通常 `-3` 选项对于应用有冲突的补丁是更加明智的选择。

```
$ git am -3 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

比如上面这种情况，我在之前已经应用过同样的补丁。如果没有 `-3` 选项的话，这看起来就像是存在一个冲突。

如果你正在利用一个 mbox 文件应用多个补丁，也可以在交互模式下运行 `am` 命令，这样在每个补丁之前，它会停住询问你是否要应用该补丁：

```
$ git am -3 -i mbox
Commit Body is:
--------------------------
seeing if this helps the gem
--------------------------
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

这在你保存的补丁较多时很好用，因为你可以在应用之前查看忘掉内容的补丁，并且跳过已经应用过的补丁。

当与你的特性相关的所有补丁都被应用并提交到分支中之后，你就可以选择是否以及如何将其整合到更长期的分支中去了。

检出远程分支

如果你的贡献者建立了自己的版本库，并且向其中推送了若干修改，之后将版本库的 URL 和包含更改的远程分支发送给你，那么你可以将其添加为一个远程分支，并且在本地进行合并。

比如 Jessica 向你发送了一封电子邮件，内容是在她的版本库中的 `ruby-client` 分支中有一个很不错的新功能，为了测试该功能，你可以将其添加为一个远程分支，并在本地检出：

```
$ git remote add jessica git://github.com/jessica/myproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

如果她再次发邮件说另一个分支中包含另一个优秀功能，因为之前已经设置好远程分支了，你就可以直接进行抓取及检出操作。

这对于与他人长期合作工作来说很有用。而对于提交补丁频率较小的贡献者，相对于每个人维护自己的服务器，不断增删远程分支的做法，使用电子邮件来接收可能会比较省时。况且你也不会想要加入数百个只提供一两个补丁的远程分支。然而，脚本和托管服务在一定程度上可以简化这些工作——这很大程度上依赖于你和你的贡献者开发的方式。

这种方式的另一种优点是你可以同时得到提交历史。虽然代码合并中可能会出现问题，但是你能获知他人的工作是基于你的历史中的具体哪一个位置；所以Git 会默认进行三方合并，不需要提供 `-3` 选项，你也不需要担心补丁是基于某个你无法访问的提交生成的。

对于非持续性的合作，如果你依然想要以这种方式拉取数据的话，你可以对远程版本库的 URL 调用 `git pull` 命令。这会执行一个一次性的抓取，而不会将该 URL 存为远程引用：

```
$ git pull https://github.com/onetimeguy/project
From https://github.com/onetimeguy/project
 * branch            HEAD       -> FETCH_HEAD
Merge made by recursive.
```

确定引入了哪些东西

你已经有了一个包含其他人贡献的特性分支。现在你可以决定如何处理它们了。本节回顾了若干命令，以便于你检查若将其合并入主分支所引入的更改。

一般来说，你应该对该分支中所有 master 分支尚未包含的提交进行检查。通过在分支名称前加入 `--not` 选项，你可以排除 master 分支中的提交。这和我们之前使用的 `master..contrib` 格式是一样的。假设贡献者向你发送了两个补丁，为此你创建了一个名叫 `contrib` 的分支并在其上应用补丁，你可以运行：

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700

    seeing if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700

    updated the gemspec to hopefully work better
```

如果要查看每次提交所引入的具体修改，你应该记得可以给 `git log` 命令传递 `-p` 选项，这样它会在每次提交后面附加对应的差异（diff）。

而要查看将该特性分支与另一个分支合并的完整 diff，你可能需要使用一个有些奇怪的技巧来得到正确的结果。你可能会想到这种方式：

```
$ git diff master
```

这个命令会输出一个 diff，但它可能并不是我们想要的。如果在你创建特性分支之后，`master` 分支向前移动了，你获得的结果就会显得有些不对。这是因为 Git 会直接将该特性分支与 `master` 分支的最新提交快照进行比较。比如说你在 `master` 分支中向某个文件添加了一行内容，那么直接比对最新快照的

结果看上去就像是你在特性分支中将这一行删除了。

如果 `master` 分支是你的特性分支的直接祖先，其实是没有任何问题的；但是一旦两个分支的历史产生了分叉，上述比对产生的 diff 看上去就像是将特性分支中所有的新东西加入，并且将 `master` 分支所独有的东西删除。

而你真正想要检查的东西，实际上仅仅是特性分支所添加的更改——也就是该分支与 master 分支合并所要引入的工作。要达到此目的，你需要让 Git 对特性分支上最新的提交与该分支与 master 分支的首个公共祖先进行比较。

从技术的角度讲，你可以以手工的方式找出公共祖先，并对其显式运行 diff 命令：

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

然而，这种做法比较麻烦，所以 Git 提供了一种比较便捷的方式：三点语法。对于 `diff` 命令来说，你可以通过把 `...` 置于另一个分支名后来对该分支的最新提交与两个分支的共同祖先进行比较：

```
$ git diff master...contrib
```

该命令仅会显示自当前特性分支与 master 分支的共同祖先起，该分支中的工作。这个语法很有用，应该牢记。

将贡献的工作整合进来

当特性分支中所有的工作都已经准备好整合进入更靠近主线的分支时，接下来的问题就是如何进行整合了。此外，还有一个问题是，你想使用怎样的总体工作流来维护你的项目？你的选择有很多，我们会介绍其中的一部分。

合并工作流

一种非常简单的工作流会直接将工作合并进入 `master` 分支。在这种情况下，`master` 分支包含的代码是基本稳定的。当你完成某个特性分支的工作，或审核通过了其他人所贡献的工作时，你会将其合并进入 master 分支，之后将特性分支删除，如此反复。如果我们的版本库包含类似 Figure 5-20 的两个名称分别为 `ruby_client` 和 `php_client` 的分支，并且我们先合并 `ruby_client` 分支，之后合并 `php_client` 分支，那么提交历史最后会变成 Figure 5-21 的样子。

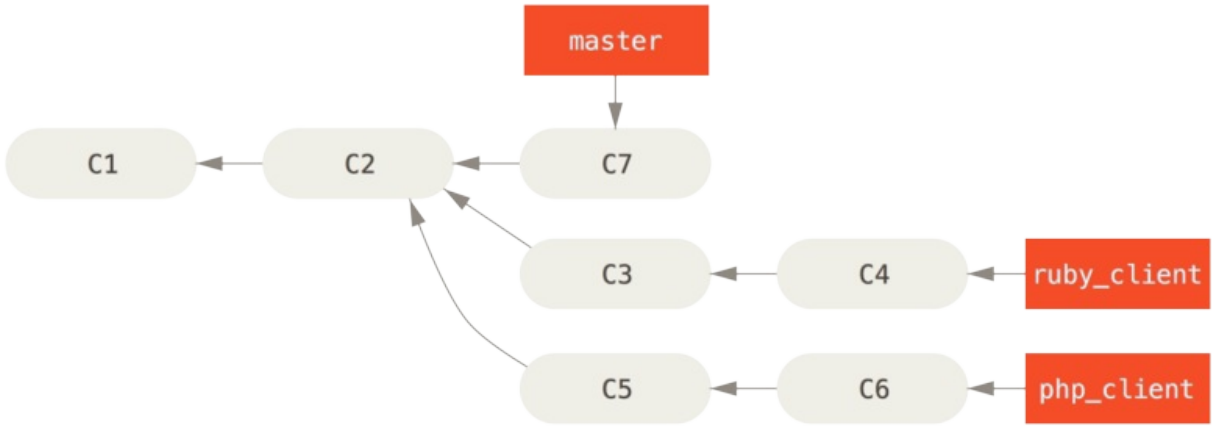


Figure 5-20. 包含若干特性分支的提交历史。

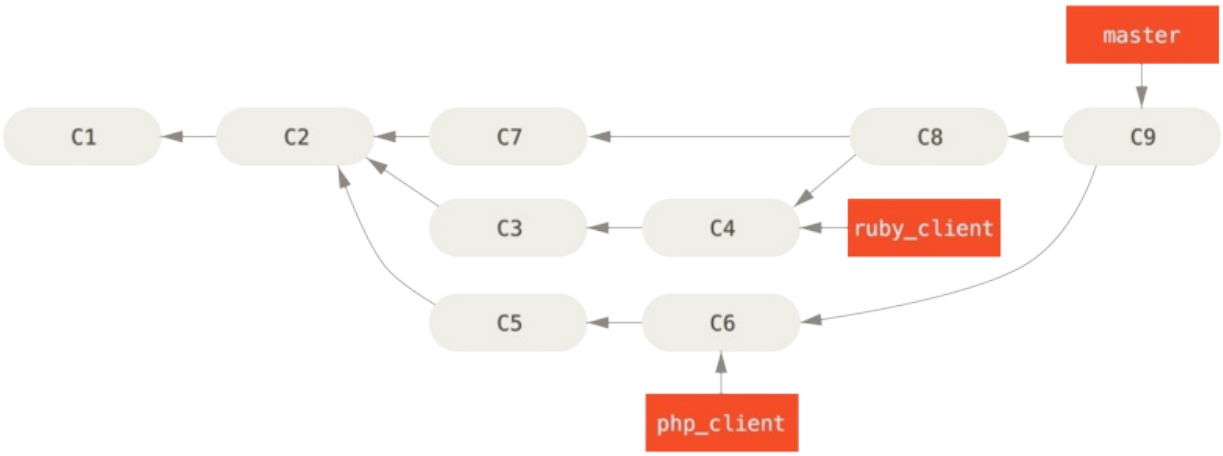


Figure 5-21. 合并特性分支之后。

这也许是最简单的工作流了，但是当项目更大，或更稳定，你对自己所引入的工作更加在意时，它可能会带来问题。

如果你的项目非常重要，你可能会使用两阶段合并循环。在这种情况下，你会维护两个长期分支，分别是 `master` 和 `develop`，`master` 分支只会在一个非常稳定的版本发布时才会更新，而所有的新代码会首先整合进入 `develop` 分支。你定期将这两个分支推送到公共版本库中。每次需要合并新的特性分支时（Figure 5-22），你都应该合并进入 `develop` 分支（Figure 5-23）；当打标签发布的时候，你会将 `master` 分支快进到已经稳定的 `develop` 分支（Figure 5-24）。

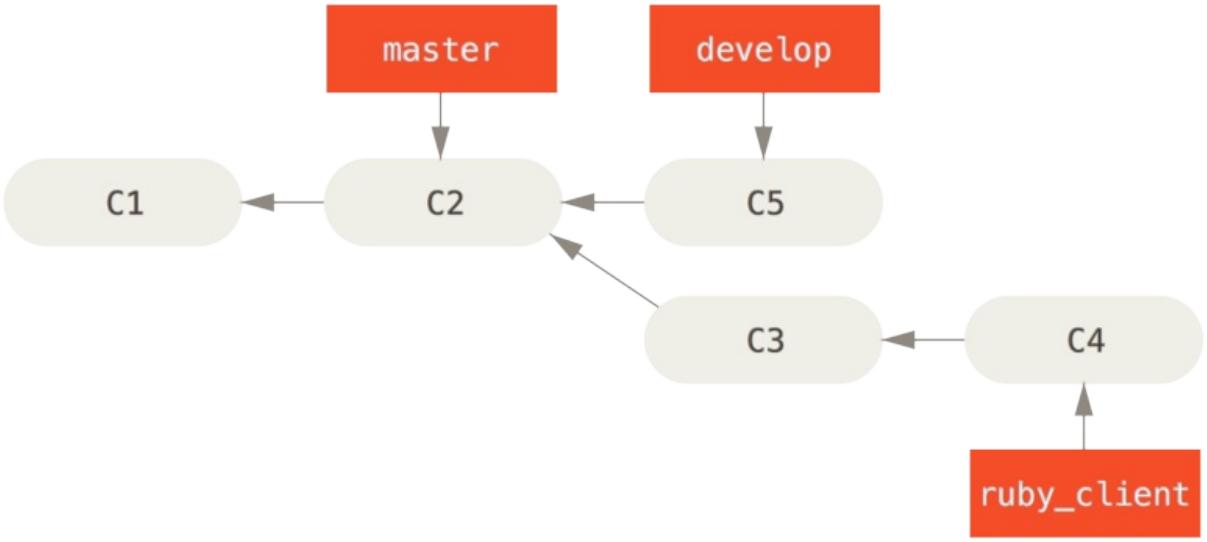


Figure 5-22. 合并特性分支前。

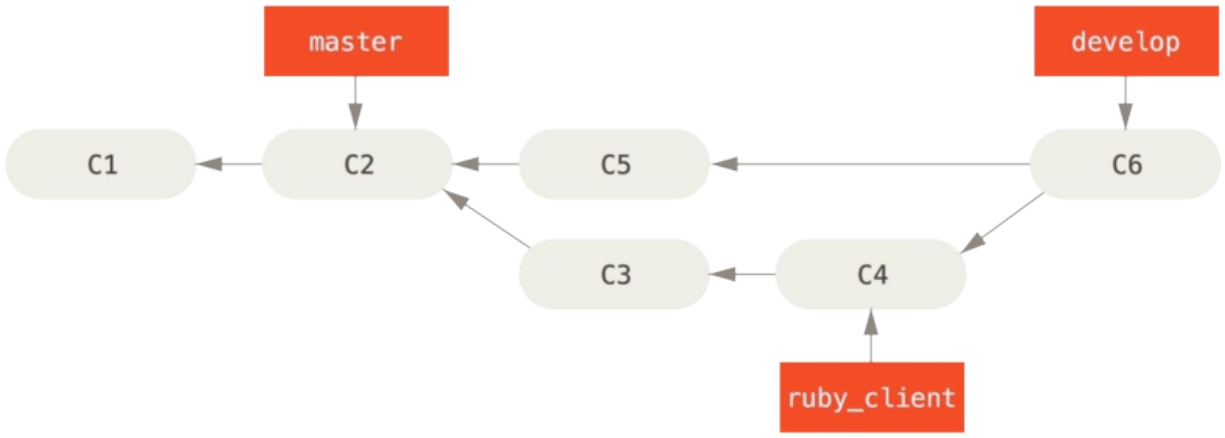


Figure 5-23. 合并特性分支后。

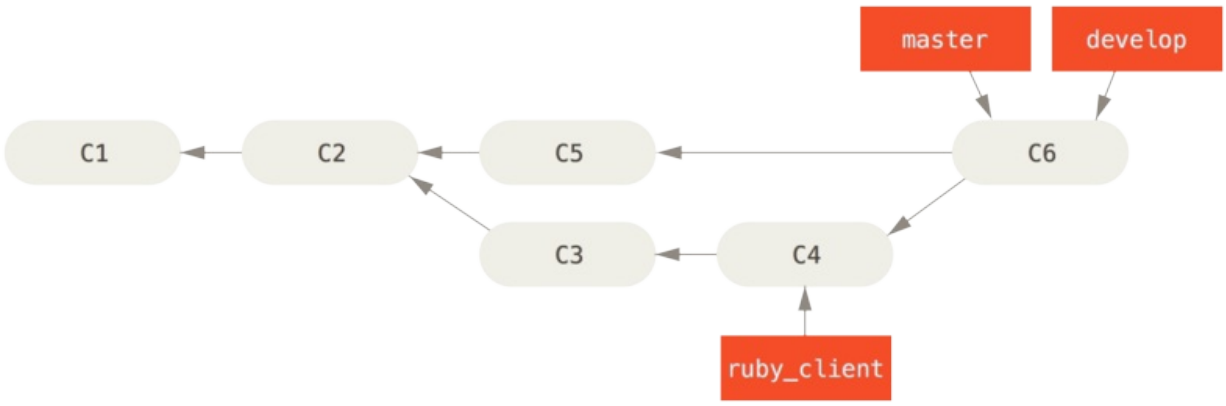


Figure 5-24. 一次发布之后。

这样当人们克隆你项目的版本库后，既可以检出 master 分支以构建最新的稳定版本并保持更新，也可以检出包含更多新东西的 develop 分支。你也可以扩展这个概念，维护一个将所有工作合并到一起的整合分支。当该分支的代码稳定并通过测试之后，将其合并进入 develop 分支；经过一段时间，确认其稳定之后，将其以快进的形式并入 master 分支。

大项目合并工作流

Git 项目包含四个长期分支：`master`、`next`，用于新工作的 `pu`（proposed updates）和用于维护性向后移植工作（maintenance backports）的 `maint` 分支。贡献者的新工作会以类似之前所介绍的方式收入特性分支中（见 Figure 5-25）。之后对特性分支进行测试评估，检查其是否已经能够合并，或者仍需要更多工作。安全的特性分支会被合并入 `next` 分支，之后该分支会被推送使得所有人都可以尝试整合到一起的特性。

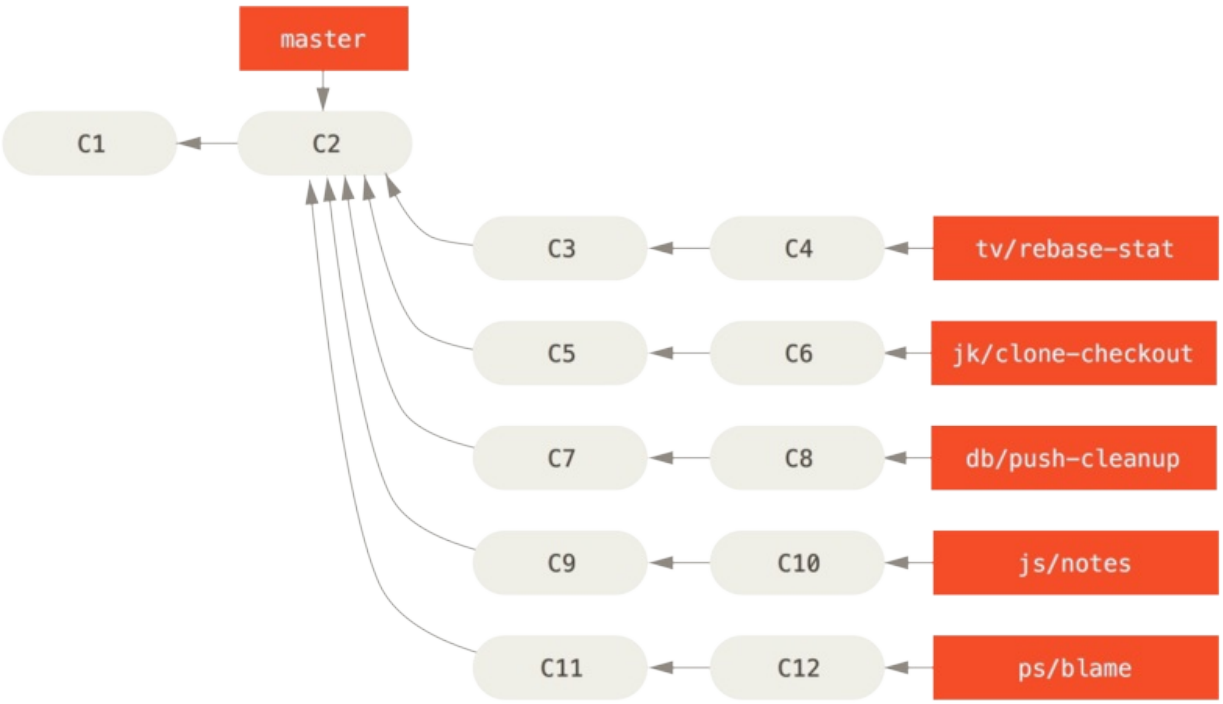


Figure 5-25. 管理复杂的一系列接收贡献的平行特性分支。

如果特性分支需要更多工作，它则会被并入 `pu` 分支。当它们完全稳定之后，会被再次并入 `master` 分支。这意味着 `master` 分支始终在进行快进，`next` 分支偶尔会被变基，而 `pu` 分支的变基比较频繁：

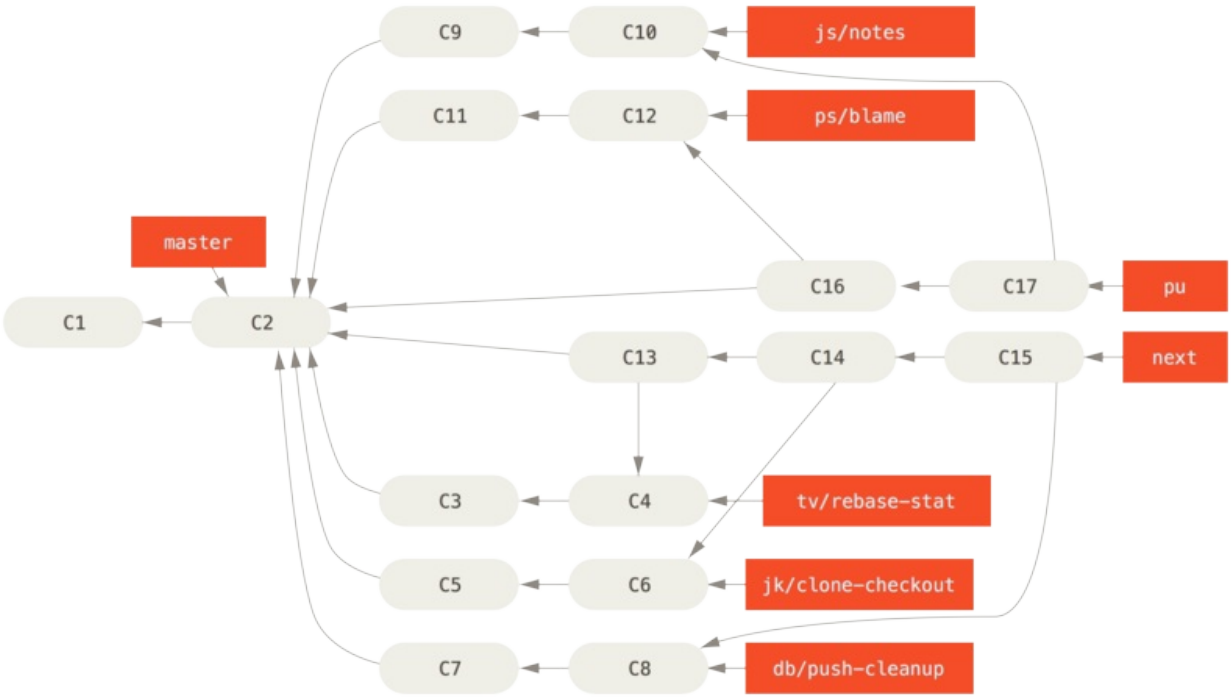


Figure 5-26. 将贡献的特性分支并入长期整合分支。

当特性分支最终被并入 `master` 分支后，便会被从版本库中删除掉。Git 项目还有一个从上一次发布中派生出来的 `maint` 分支来提供向后移植过来的补丁以供发布维护更新。因此，当你克隆 Git 的版本库之后，就会有四个可分别评估该项目开发的不同阶段的可检出的分支，检出哪个分支，取决于你需要多新的版本，或者你想要如何进行贡献；对于维护者来说，这套结构化的工作流能帮助它们审查新的贡献。

变基与拣选工作流

为了保持线性的提交历史，有些维护者更喜欢在 master 分支上对贡献过来的工作进行变基和拣选，而不是直接将其合并。当你完成了某个特性分支中的工作，并且决定要将其整合的时候，你可以在该分支中运行变基命令，在当前 master 分支（或者是 `develop` 等分支）的基础上重新构造修改。如果结果理想的话，你可以快进 `master` 分支，最后得到一个线性的项目提交历史。

另一种将引入的工作转移到其他分支的方法是拣选。Git 中的拣选类似于对特定的某次提交的变基。它会提取该提交的补丁，之后尝试将其重新应用到当前分支上。这种方式在你只想引入特性分支中的某个提交，或者特性分支中只有一个提交，而你不想运行变基时很有用。举个例子，假设你的项目提交历史类似：

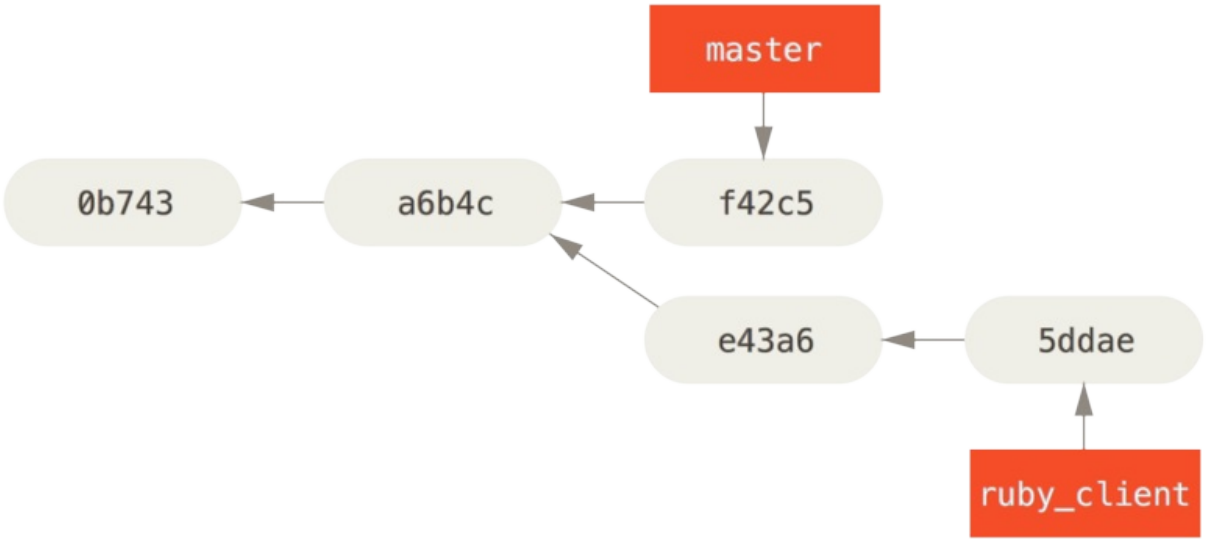


Figure 5-27. 拣选之前的示例历史。

如果你希望将提交 `e43a6` 拉取到 master 分支，你可以运行：

```
$ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcdf
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
 3 files changed, 17 insertions(+), 3 deletions(-)
```

这样会拉取和 `e43a6` 相同的更改，但是因为应用的日期不同，你会得到一个新的提交 SHA-1 值。现在你的历史会变成这样：

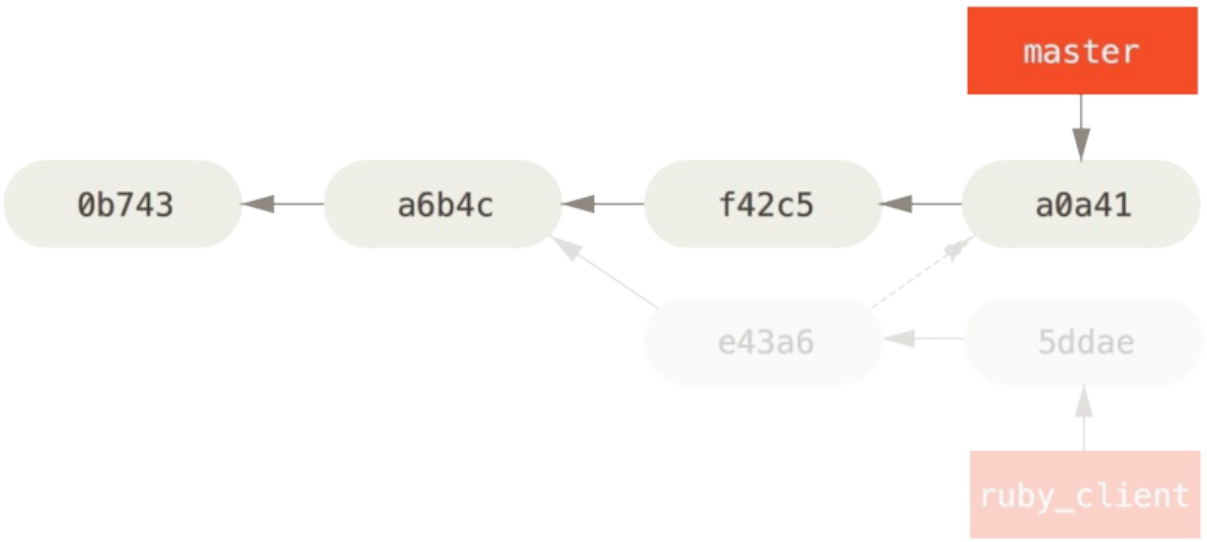


Figure 5-28. 拣选特性分支中的一个提交后的历史。

现在你可以删除这个特性分支，并丢弃不想拉入的提交。

Rerere

如果你在进行大量的合并或变基，或维护一个长期的特性分支，Git 提供的一个叫做“rerere”的功能会有一些帮助。

Rerere 是“重用已记录的冲突解决方案（reuse recorded resolution）”的意思——它是一种简化冲突解决的方法。当启用 rerere 时，Git 将会维护一些成功合并之前和之后的镜像，当 Git 发现之前已经修复过类似的冲突时，便会使用之前的修复方案，而不需要你的干预。

这个功能包含两个部分：一个配置选项和一个命令。其中的配置选项是 `rerere.enabled`，把它放在全局配置中就可以了：

```
$ git config --global rerere.enabled true
```

现在每当你进行一次需要解决冲突的合并时，解决方案都会被记录在缓存中，以备之后使用。

如果你需要和 rerere 的缓存交互，你可以使用 `git rerere` 命令。当单独调用它时，Git 会检查解决方案数据库，尝试寻找一个和当前任一冲突相关的匹配项并解决冲突（尽管当 `rerere.enabled` 被设置为 `true` 时会自动进行）。它也有若干子命令，可用来查看记录项，删除特定解决方案和清除缓存全部内容等。我们将在 “Rerere” 中详细探讨。

为发布打标签

当你决定进行一次发布时，你可能想要留下一个标签，这样在之后的任何一个提交点都可以重新创建该发布。你在 Chapter 2 中已经了解了创建新标签的过程。作为一个维护者，如果你决定要为标签签名的话，打标签的过程应该是这样子的：

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon <schacon@gmail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

如果你为标签签名了，你可能会遇到分发用来签名的 PGP 公钥的问题。Git 项目的维护者已经解决了这一问题，其方法是在版本库中以 blob 对象的形式包含他们的公钥，并添加一个直接指向该内容的标签。要完成这一任务，首先你可以通过运行 `gpg --list-keys` 找出你所想要的 key：

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
---------------------------------
pub   1024D/F721C45A 2009-02-09 [expires: 2010-02-09]
uid                  Scott Chacon <schacon@gmail.com>
sub   2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

之后你可以通过导出 key 并通过管道传递给 `git hash-object` 来直接将 key 导入到 Git 的数据库中，`git hash-object` 命令会向 Git 中写入一个包含其内容的新 blob 对象，并向你返回该 blob 对象的 SHA-1 值：

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

既然 Git 中已经包含你的 key 的内容了，你就可以通过指定由 `hash-object` 命令给出的新 SHA-1 值来创建一个直接指向它的标签：

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

如果你运行 `git push --tags` 命令，那么 `maintainer-pgp-pub` 标签将会被共享给所有人。需要校验标签的人可以通过从数据库中直接拉取 blob 对象并导入到 GPG 中来导入 PGP key：

```
$ git show maintainer-pgp-pub | gpg --import
```

人们可以使用这个 key 来校验所有由你签名的标签。另外，如果你在标签信息中包含了一些操作说明，用户可以通过运行 `git show <tag>` 来获取更多关于标签校验的说明。

生成一个构建号

Git 中不存在随每次提交递增的“v123”之类的数字序列，如果你想要为提交附上一个可读的名称，可以对其运行 `git describe` 命令。Git 将会给出一个字符串，它由最近的标签名、自该标签之后的提交数目和你所描述的提交的部分 SHA-1 值构成：

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

这样你在导出一个快照或构建时，可以给出一个便于人们理解的命名。实际上，如果你的 Git 是从 Git 自己的版本库克隆下来并构建的，那么 `git --version` 命令给出的结果是与此类似的。如果你所描述的提交自身就有一个标签，那么它将只会输出标签名，没有后面两项信息。

注意 `git describe` 命令只适用于有注解的标签（即使用 `-a` 或 `-s` 选项创建的标签），所以如果你在使用 `git describe` 命令的话，为了确保能为标签生成合适的名称，打发布标签时都应该采用加注解的方式。你也可以使用这个字符串来调用 checkout 或 show 命令，但是这依赖于其末尾的简短 SHA-1 值，因此不一定一直有效。比如，最近 Linux 内核为了保证 SHA-1 值对象的唯一性，将其位数由 8 位扩展到了 10 位，导致以前的 `git describe` 输出全部失效。

准备一次发布

现在你可以发布一个构建了。其中一件事情就是为那些不使用 Git 的可怜包们创建一个最新的快照归档。使用 `git archive` 命令完成此工作：

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

如果有人将这个压缩包解压，他就可以得到你的项目文件夹的最新快照。你也可以以类似的方式创建一个 zip 压缩包，但此时你应该向 `git archive` 命令传递 `--format=zip` 选项：

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

现在你有了本次发布的一个 tar 包和一个 zip 包，可以将其上传到网站或以电子邮件的形式发送给人们。

制作提交简报

现在是时候通知邮件列表里那些好奇你的项目发生了什么的人了。使用 `git shortlog` 命令可以快速生成一份包含从上次发布之后项目新增内容的修改日志（changelog）类文档。它会对你给定范围内的所有提交进行总结；比如，你的上一次发布名称是 v1.0.1，那么下面的命令可以给出上次发布以来所有提交的总结：

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (8):
      Add support for annotated tags to Grit::Tag
      Add packed-refs annotated tag support.
      Add Grit::Commit#to_patch
      Update version and History.txt
      Remove stray `puts`
      Make ls_tree ignore nils

Tom Preston-Werner (4):
      fix dates in history
      dynamic version method
      Version bump to 1.0.2
      Regenerated gemspec for version 1.0.2
```

这份整洁的总结包括了自 v1.0.1 以来的所有提交，并且已经按照作者分好组，你可以通过电子邮件将其直接发送到列表中。

5.4 总结

总结

你现在能自如地使用 Git 为项目做出贡献、维护自己的项目或采纳其他用户的贡献了。恭喜你成为了一个高效的 Git 开发者！下一章中，你将会学到如何使用规模最大最流行的 Git 托管服务，GitHub。

6. GitHub

第6章. GitHub

GitHub 是最大的 Git 版本库托管商，是成千上万的开发者和项目能够合作进行的中心。大部分 Git 版本库都托管在 GitHub，很多开源项目使用 GitHub 实现 Git 托管、问题追踪、代码审查以及其它事情。所以，尽管这不是 Git 开源项目的直接部分，但如果想要专业地使用 Git，你将不可避免地与 GitHub 打交道，所以这依然是一个绝好的学习机会。

本章将讨论如何高效地使用 GitHub。我们将学习如何注册和管理账户、创建和使用 Git 版本库、向已有项目贡献的通用流程以及如何接受别人向你自己项目的贡献、GitHub 的编程接口和很多能够让这些操作更简单的小提示。

如果你对如何使用 GitHub 托管自己的项目，或者与已经托管在 GitHub 上面的项目进行合作没有兴趣，可以直接跳到 Chapter 7 这一章。

6.1 账户的创建和配置

账户的创建和配置

你所需要做的第一件事是创建一个免费账户。直接访问 https://github.com，选择一个未被占用的用户名，提供一个电子邮件地址和密码，点击写着“Sign up for GitHub”的绿色大按钮即可。

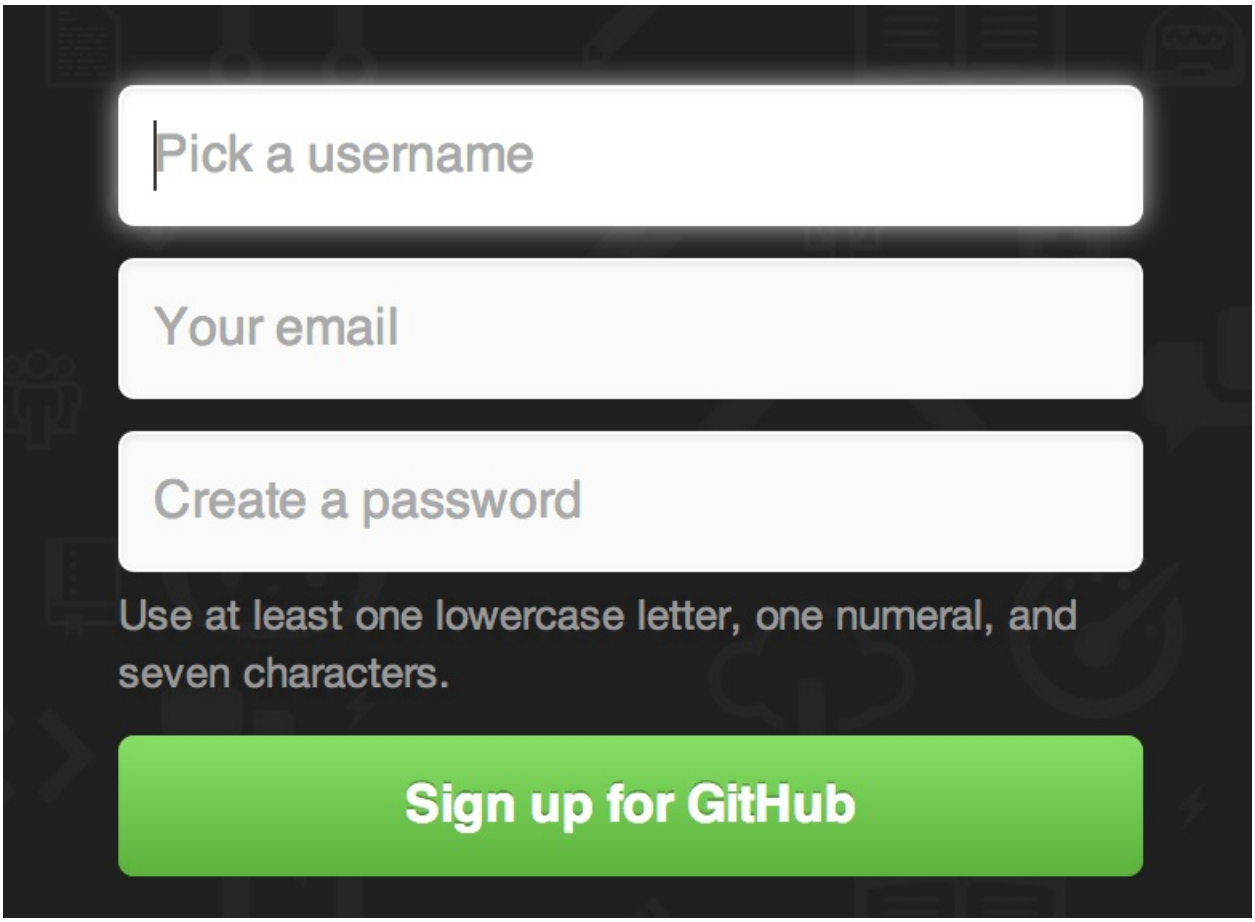


Figure 6-1. GitHub 注册表单。

你将看到的下一个页面是升级计划的价格页面，目前我们可以直接忽略这个页面。GitHub 会给你提供的邮件地址发送一封验证邮件。尽快到你的邮箱进行验证，这是非常重要的（我们会在后面了解到这点）。

GitHub 为免费账户提供了完整功能，限制是你的项目都将被完全公开（每个人都具有读权限）。GitHub 的付费计划可以让你拥有一定数目的私有项目，不过本书将不涉及这部分内容。

点击屏幕左上角的 Octocat 图标，你将来到控制面板页面。现在，你已经做好了使用 GitHub 的准备工作。

SSH 访问

现在，你完全可以使用 `https://` 协议，通过你刚刚创建的用户名和密码访问 Git 版本库。但是，如果仅仅克隆公有项目，你甚至不需要注册——刚刚我们创建的账户是为了以后 fork 其它项目，以及推送我们自己的修改。

如果你习惯使用 SSH 远程，你需要配置一个公钥。（如果你没有公钥，参考 “生成 SSH 公钥” 。）使用窗口右上角的链接打开你的账户设置：

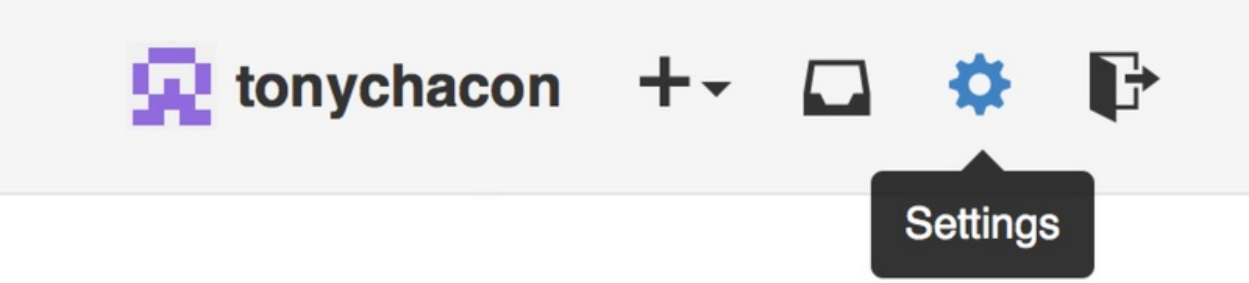


Figure 6-2. “Account settings”链接。

然后在左侧选择“SSH keys”部分。

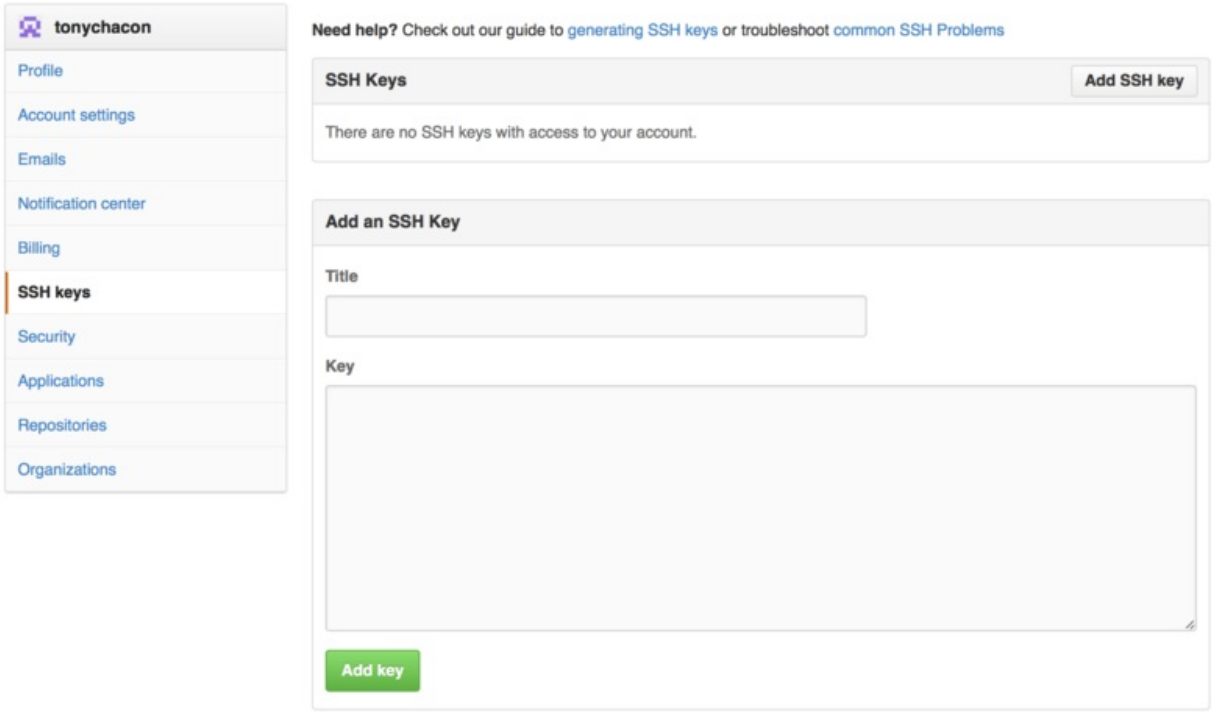


Figure 6-3. “SSH keys”链接。

在这个页面点击“`Add an SSH key`”按钮，给你的公钥起一个名字，将你的 `~/.ssh/id_rsa.pub`（或者自定义的其它名字）公钥文件的内容粘贴到文本区，然后点击“Add key”。

确保给你的 SSH 密钥起一个能够记得住的名字。你可以为每一个密钥起名字（例如，“我的笔记本电脑”或者“工作账户”等），以便以后需要吊销密钥时能够方便地区分。

头像

下一步，如果愿意的话，你可以将生成的头像换成你喜欢的图片。首先，来到“Profile”标签页（在“SSH Keys”标签页上方），点击“Upload new picture”。

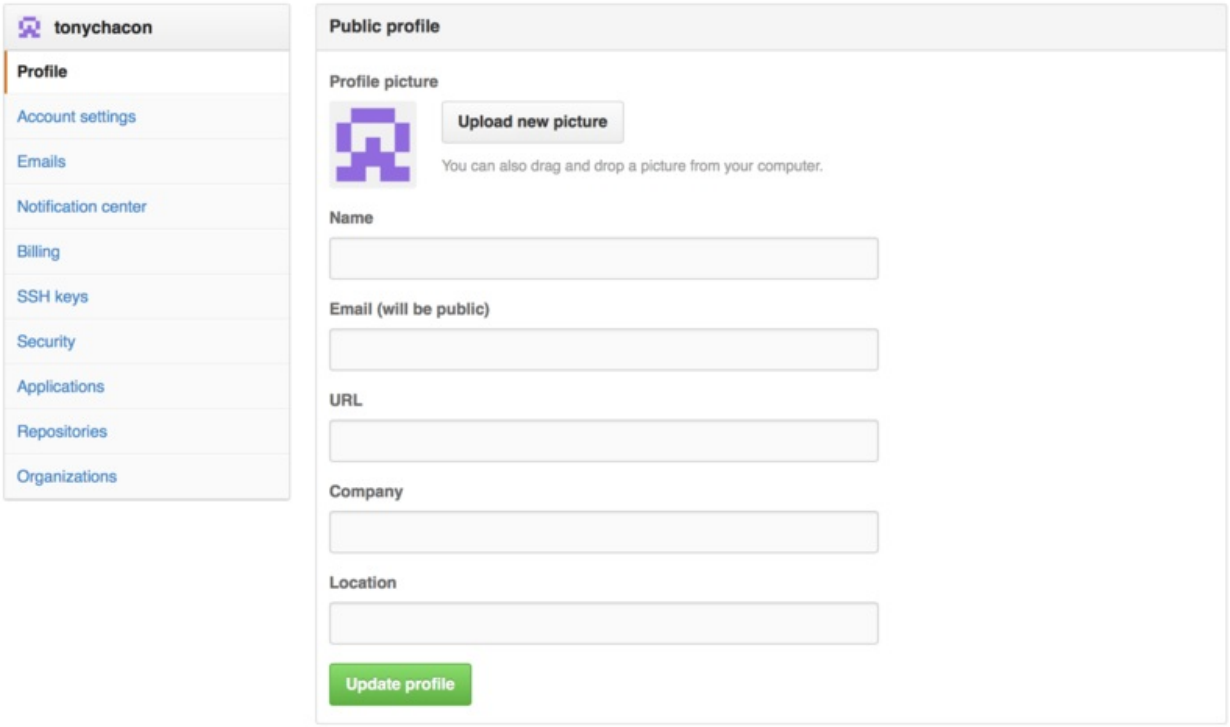


Figure 6-4. “Profile”链接。

我们选择了本地磁盘上的一个 Git 图标，上传之后还可以对其进行裁剪。

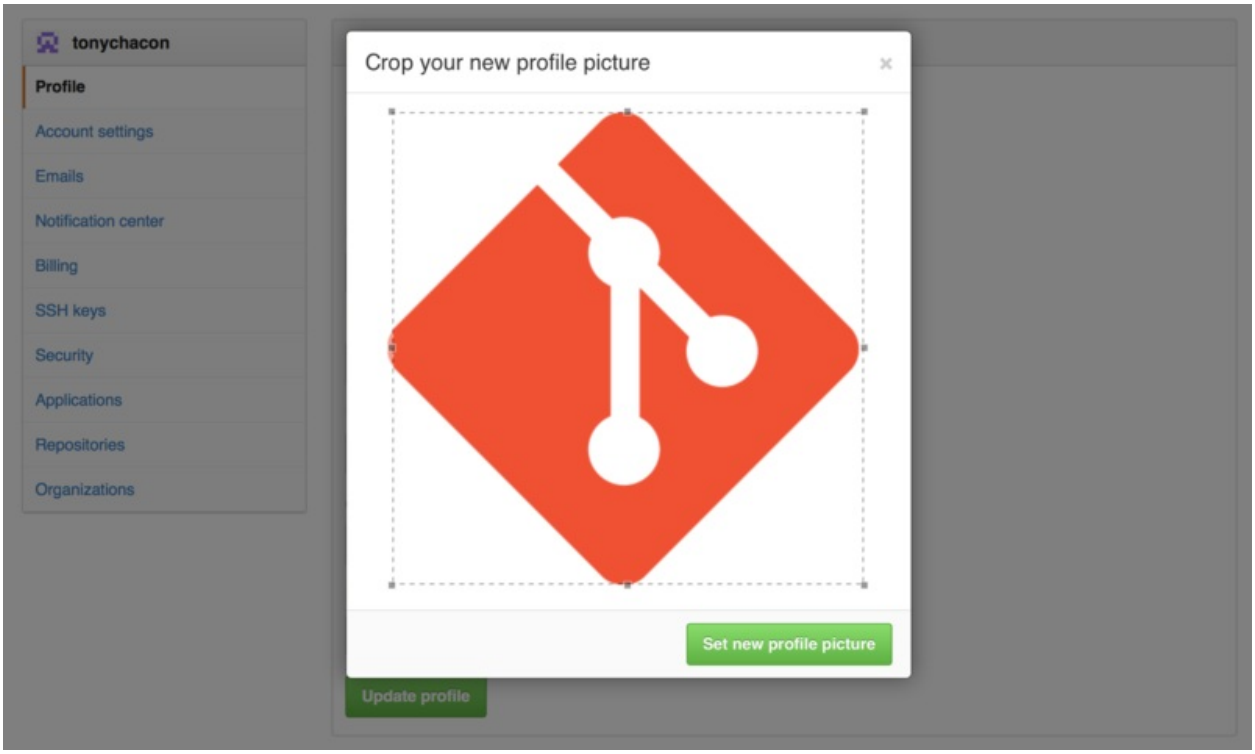


Figure 6-5. 裁剪头像

现在，在网站任意有你参与的位置，人们都可以在你的用户名旁边看到你的头像。

如果你已经把头像上传到了流行的 Gravatar 托管服务（Wordpress 账户经常使用），默认就会使用这个头像，因此，你就不需要进行这一步骤了。

邮件地址

GitHub 使用用户邮件地址区分 Git 提交。如果你在自己的提交中使用了多个邮件地址，希望 GitHub 可以正确地将它们连接起来，你需要在管理页面的 Emails 部分添加你拥有的所有邮箱地址。

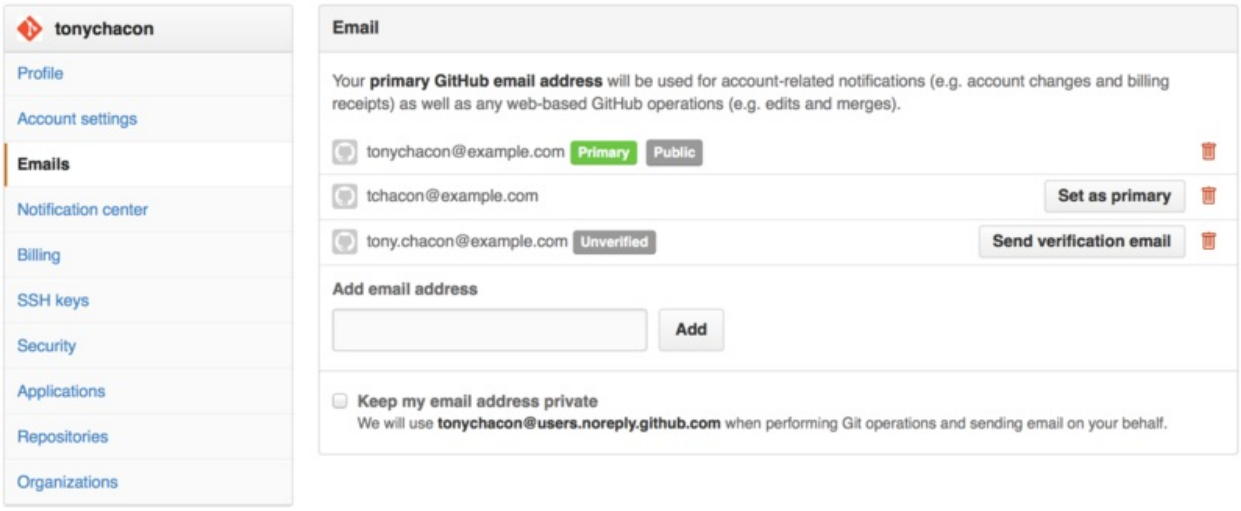


Figure 6-6. 添加邮件地址

在 Figure 6-6 中我们可以看到一些不同的状态。顶部的地址是通过验证的，并且被设置为主要地址，这意味着该地址会接收到所有的通知和回复。第二个地址是通过验证的，如果愿意的话，可以将其设置为主要地址。最后一个地址是未通过验证的，这意味着你不能将其设置为主要地址。当 GitHub 发现任意版本库中的任意提交信息包含了这些地址，它就会将其链接到你的账户。

两步验证

最后，为了额外的安全性，你绝对应当设置两步验证，简写为 “2FA” 。两步验证是一种用于降低因你的密码被盗而带来的账户风险的验证机制，现在已经变得越来越流行。开启两步验证，GitHub 会要求你用两种不同的验证方法，这样，即使其中一个被攻破，攻击者也不能访问你的账户。

你可以在 Account settings 页面的 Security 标签页中找到 Two-factor Authentication 设置。

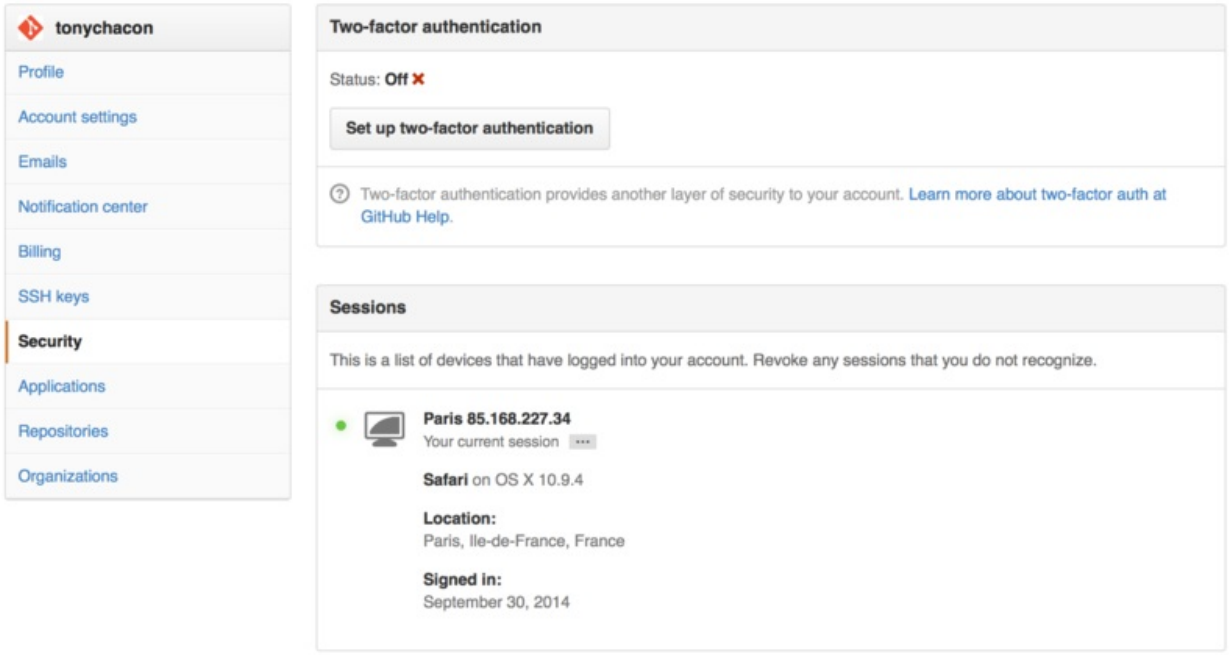


Figure 6-7. Security 标签页中的 2FA

点击“Set up two-factor authentication”按钮，会跳转到设置页面。该页面允许你选择是要在登录时使用手机 app 生成辅助码（一种“基于时间的一次性密码”），还是要 GitHub 通过 SMS 发送辅助码。

选择合适的方法后，按照提示步骤设置 2FA，你的账户会变得更安全，每次登录 GitHub 时都需要提供除密码以外的辅助码。

6.2 对项目做出贡献

对项目做出贡献

账户已经建立好了，现在我们来了解一些能帮助你对现有的项目做出贡献的知识。

派生（Fork）项目

如果你想要参与某个项目，但是并没有推送权限，这时可以对这个项目进行“派生”。派生的意思是指，GitHub 将在你的空间中创建一个完全属于你的项目副本，且你对其具有推送权限。

> 在以前，“fork”是一个贬义词，指的是某个人使开源项目向不同的方向发展，或者创建一个竞争项目，使得原项目的贡献者分裂。在 GitHub，“fork”指的是你自己的空间中创建的项目副本，这个副本允许你以一种更开放的方式对其进行修改。

通过这种方式，项目的管理者不再需要忙着把用户添加到贡献者列表并给予他们推送权限。人们可以派生这个项目，将修改推送到派生出的项目副本中，并通过创建合并请求（Pull Request）来让他们的改动进入源版本库，下文我们会详细说明。创建了合并请求后，就会开启一个可供审查代码的板块，项目的拥有者和贡献者可以在此讨论相关修改，直到项目拥有者对其感到满意，并且认为这些修改可以被合并到版本库。

你可以通过点击项目页面右上角的“Fork”按钮，来派生这个项目。



Figure 6-8. “Fork”按钮

稍等片刻，你将被转到新项目页面，该项目包含可写的代码副本。

GitHub 流程

GitHub 设计了一个以合并请求为中心的特殊合作流程。它基于我们在 Chapter 3 的 “特性分支” 中提到的工作流程。不管你是在一个紧密的团队中使用单独的版本库，或者使用许多的“Fork”来为一个由陌生人组成的国际企业或网络做出贡献，这种合作流程都能应付。

流程通常如下：

1. 从 `master` 分支中创建一个新分支
2. 提交一些修改来改进项目
3. 将这个分支推送到 GitHub 上

4. 创建一个合并请求
5. 讨论，根据实际情况继续修改
6. 项目的拥有者合并或关闭你的合并请求

这基本和 “集成管理者工作流” 中的一体化管理流程差不多，但是团队可以使用 GitHub 提供的网页工具替代电子邮件来交流和审查修改。

现在我们来看一个使用这个流程的例子。

创建合并请求

Tony 在找一些能在他的 Arduino 微控制器上运行的代码，他觉得 https://github.com/schacon/blink 中的代码不错。

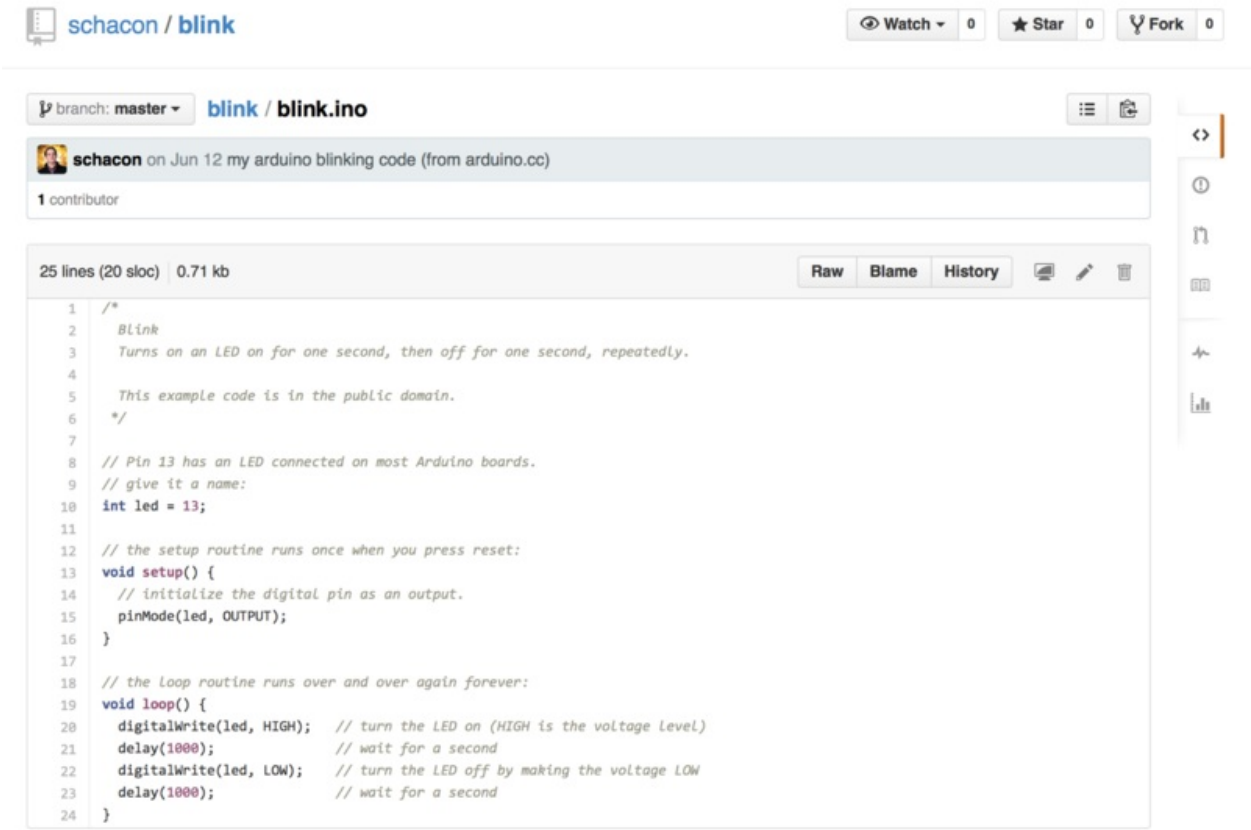


Figure 6-9. 他想要做出贡献的项目

但是有个问题，这个代码中的的闪烁频率太高，我们觉得 3 秒一次比 1 秒一次更好一些。所以让我们来改进这个程序，并将修改后的代码提交给这个项目。

首先，单击“Fork”按钮来获得这个项目的副本。我们使用的用户名是“tonychacon”，所以这个项目副本的访问地址是：`https://github.com/tonychacon/blink`。我们将它克隆到本地，创建一个

分支，修改代码，最后再将改动推送到 GitHub。

```
$ git clone https://github.com/tonychacon/blink
Cloning into 'blink'...

$ cd blink
$ git checkout -b slow-blink
Switched to a new branch 'slow-blink'

$ sed -i '' 's/1000/3000/' blink.ino

$ git diff --word-diff
diff --git a/blink.ino b/blink.ino
index 15b9911..a6cc5a5 100644
--- a/blink.ino
+++ b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  [-delay(1000);-]{+delay(3000);+}               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  [-delay(1000);-]{+delay(3000);+}               // wait for a second
}

$ git commit -a -m 'three seconds is better'
[slow-blink 5ca509d] three seconds is better
 1 file changed, 2 insertions(+), 2 deletions(-)

$ git push origin slow-blink
Username for 'https://github.com': tonychacon
Password for 'https://tonychacon@github.com':
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/tonychacon/blink
 * [new branch]      slow-blink -> slow-blink
```

❶ 将派生出的副本克隆到本地

❷ 创建出名称有意义的分支

❸ 修改代码

❹ 检查改动

❺ 将改动提交到分支中

❻ 将新分支推送到 GitHub 的副本中

现在到 GitHub 上查看之前的项目副本，可以看到 GitHub 提示我们有新的分支，并且显示了一个大大的绿色按钮让我们可以检查我们的改动，并给源项目创建合并请求。

你也可以到“Branches”（分支）页面查看分支并创建合并请求：
`https://github.com/<用户名>/<项目名>/branches`

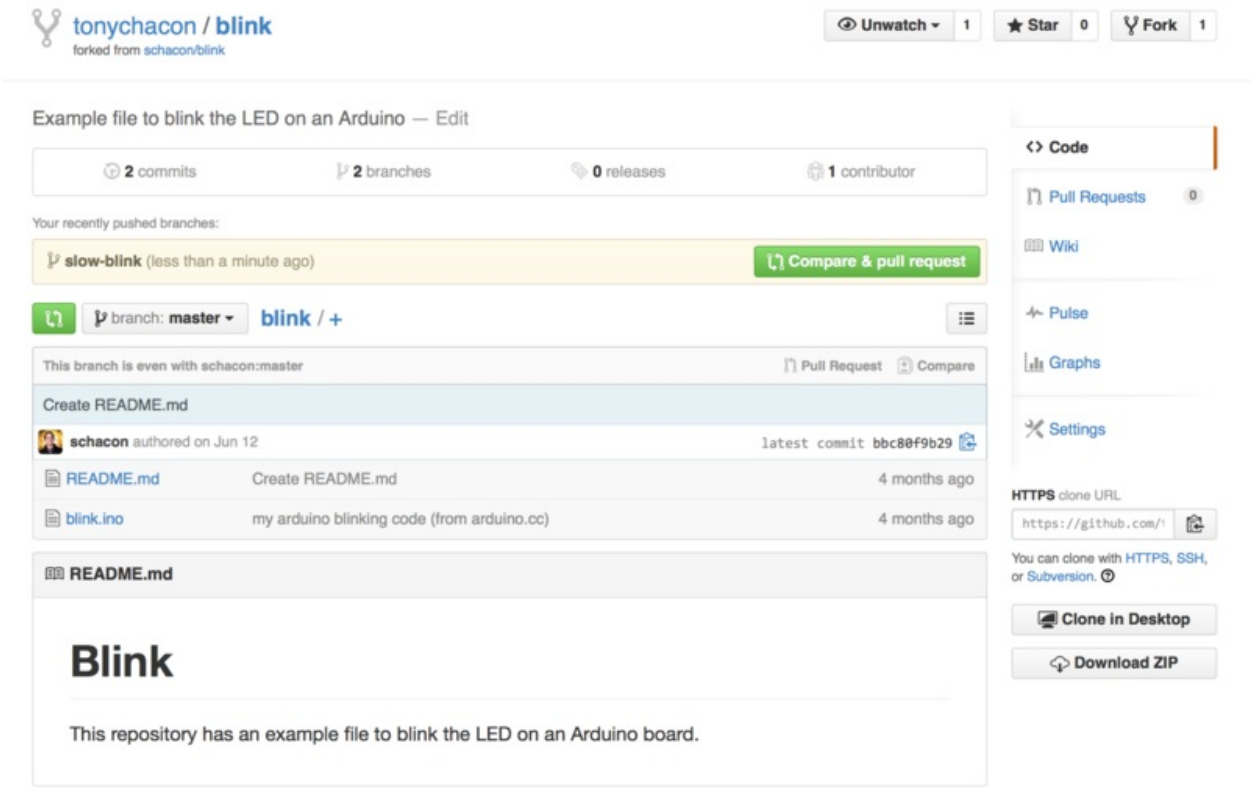


Figure 6-10. 合并请求按钮

如果你点击了那个绿色按钮，就会看到一个新页面，在这里我们可以对改动填写标题和描述，让项目的拥有者考虑一下我们的改动。通常花点时间来编写个清晰有用的描述是个不错的主意，这能让作者明白为什么这个改动可以给他的项目带来好处，并且让他接受合并请求。

同时我们也能看到比主分支中所“领先”（ahead）的提交（在这个例子中只有一个）以及所有将会被合并的改动与之前代码的对比。

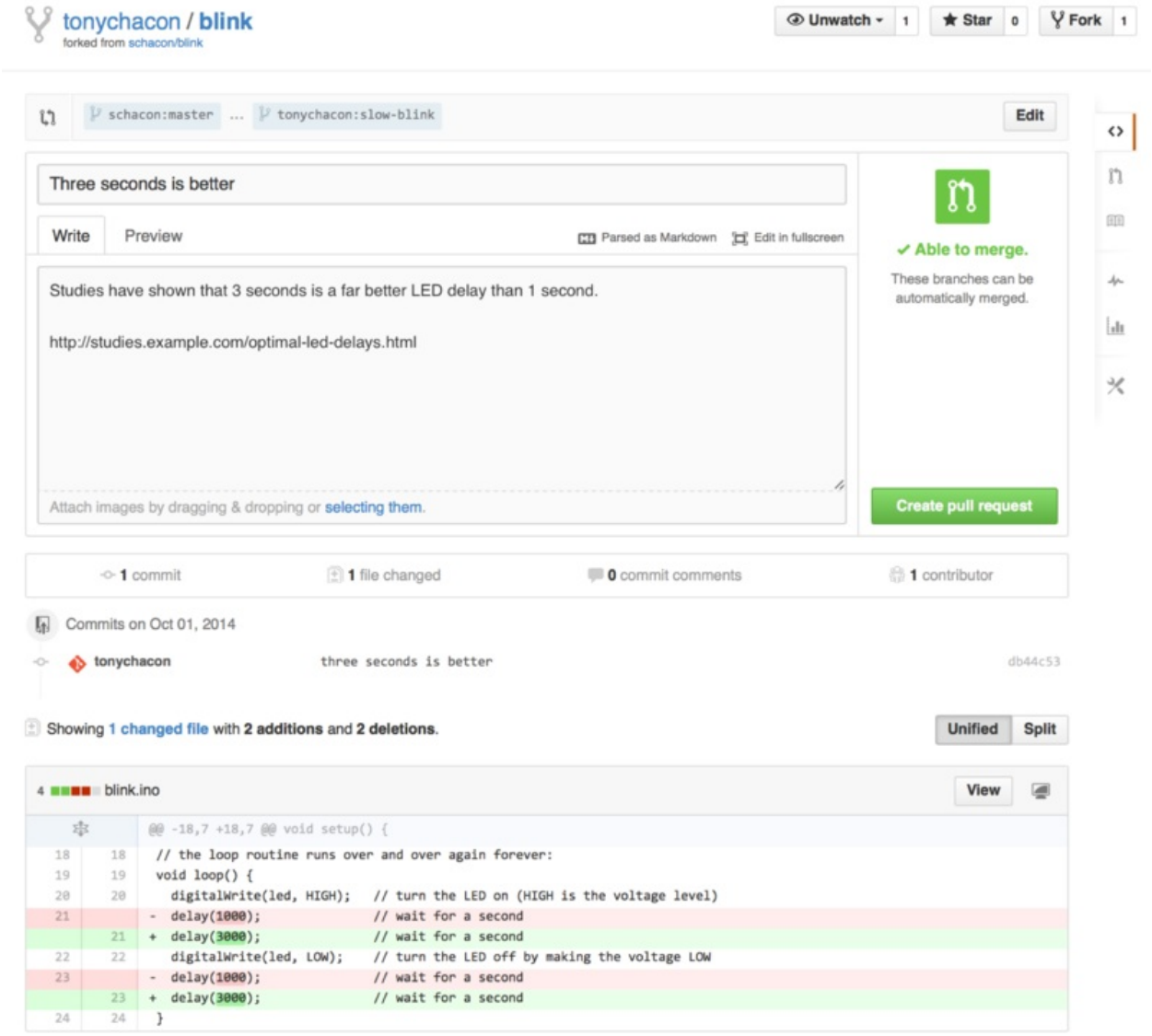


Figure 6-11. 合并请求创建页面

当你单击了“Create pull request”（创建合并请求）的按钮后，这个项目的拥有者将会收到一条包含关改动和合并请求页面的链接的提醒。

虽然合并请求通常是在贡献者准备好在公开项目中提交改动的时候提交，但是也常被用在仍处于开发阶段的内部项目中。因为合并请求在提交后 依然可以加入新的改动 ，它也经常被用来建立团队合作的环境，而不只是在最终阶段使用。

利用合并请求

现在，项目的拥有者可以看到你的改动并合并它，拒绝它或是发表评论。在这里我们就当作他喜欢这个点子，但是他想要让灯熄灭的时间比点亮的时间稍长一些。

接下来可能会通过电子邮件进行互动，就像我们在 Chapter 5 中提到的工作流程那样，但是在 GitHub，

这些都在线上完成。项目的拥有者可以审查修改，只需要单击某一行，就可以对其发表评论。

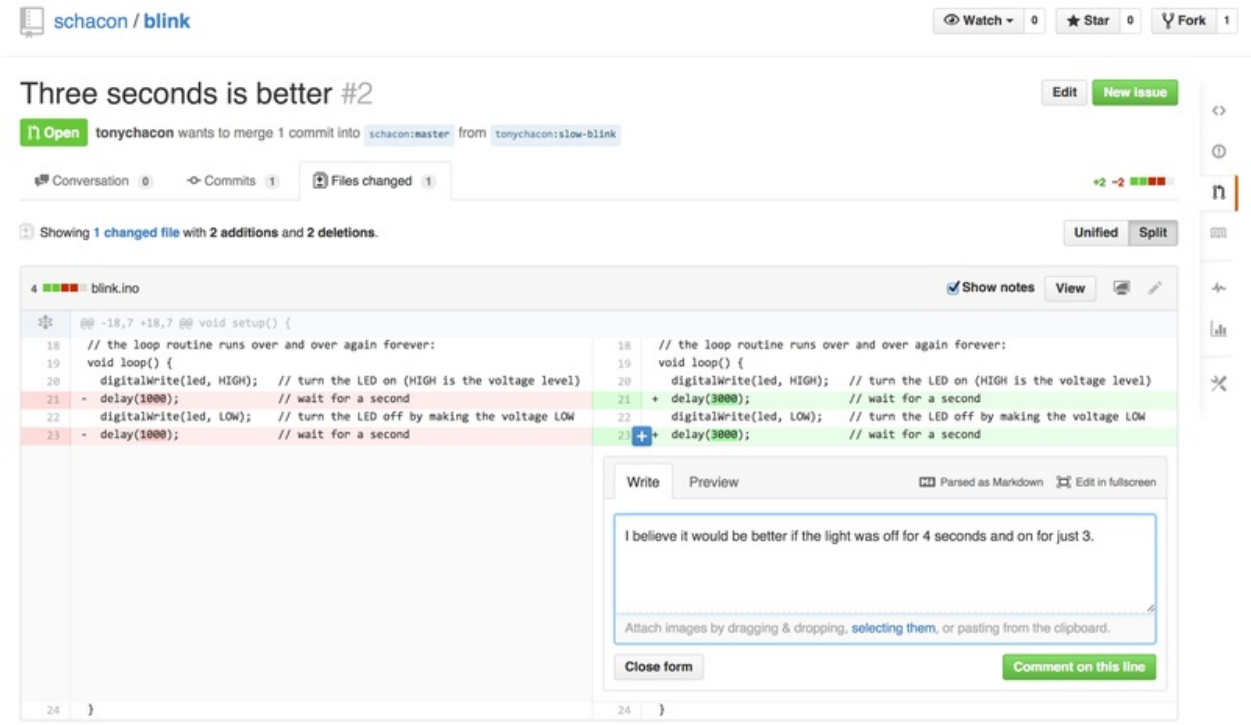


Figure 6-12. 对合并请求内的特定一行发表评论

当维护者发表评论后，提交合并请求的人，以及所有正在关注（Watching）这个版本库的用户都会收到通知。我们待会儿将会告诉你如何修改这项设置。现在，如果 Tony 有开启电子邮件提醒，他将会收到这样的一封邮件：



Figure 6-13. 通过电子邮件发送的评论提醒

每个人都能在合并请求中发表评论。在 Figure 6-14 里我们可以看到项目拥有者对某行代码发表评论，并在讨论区留下了一个普通评论。你可以看到被评论的代码也会在互动中显示出来。

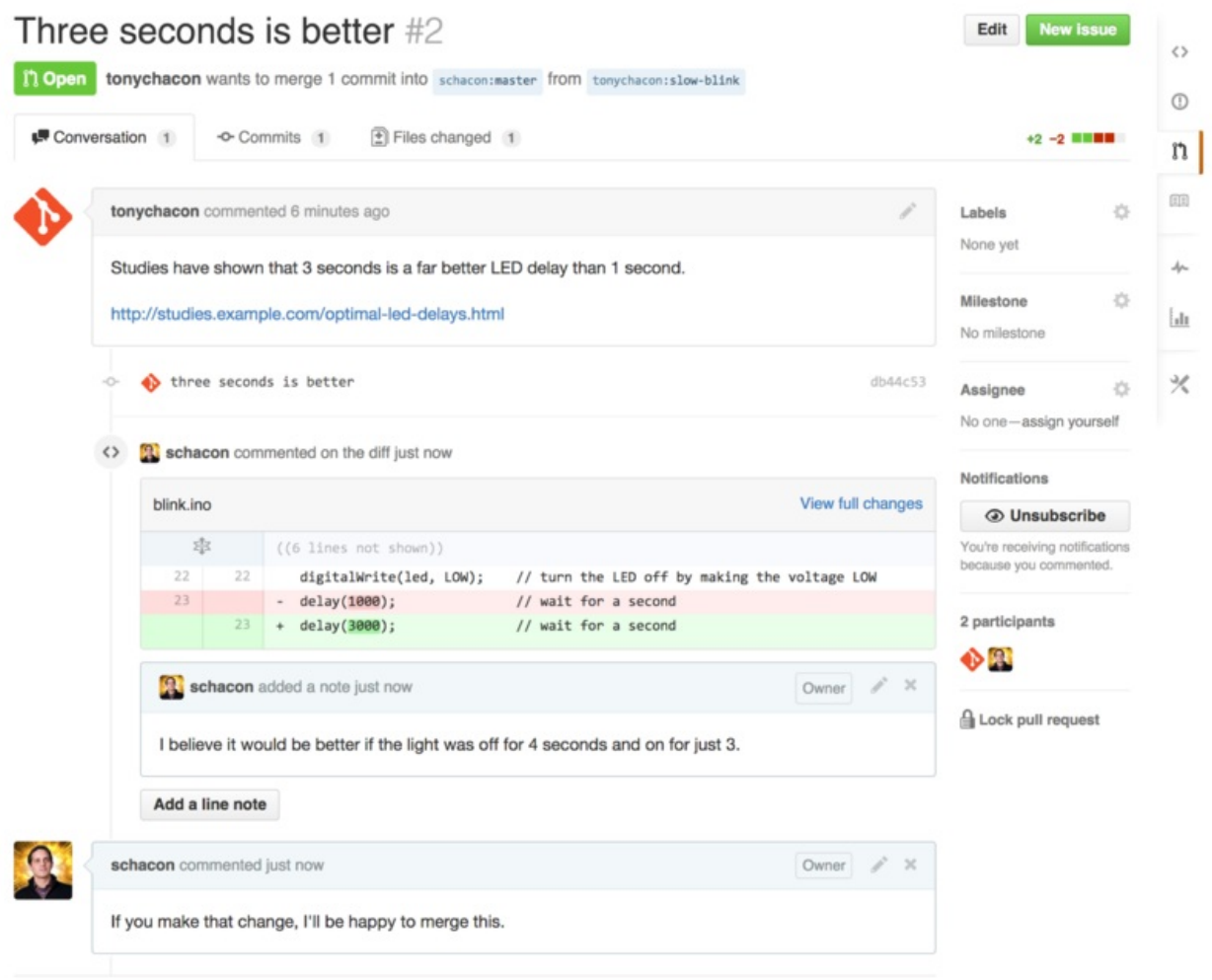


Figure 6-14. 合并请求讨论页面

现在贡献者可以看到如何做才能让他们的改动被接受。幸运的是，这也是一件轻松的事情。如果你使用的是电子邮件进行交流，你需要再次对代码进行修改并重新提交至邮件列表，在 GitHub 上，你只需要再次提交到你的分支中并推送即可。

如果贡献者完成了以上的操作，项目的拥有者会再次收到提醒，当他们查看页面时，将会看到最新的改动。事实上，只要提交中有一行代码改动，GitHub 都会注意到并处理掉旧的变更集。

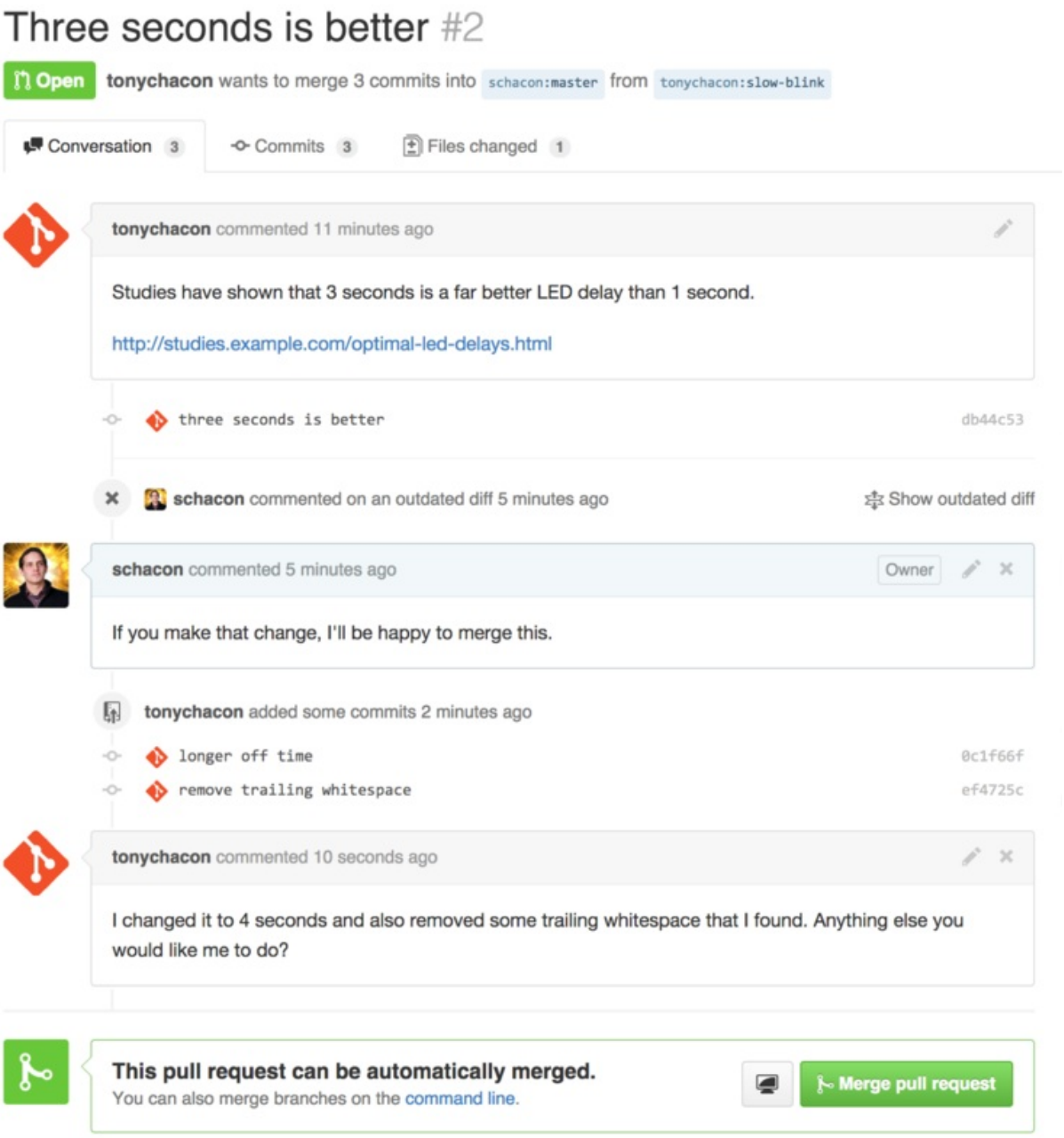


Figure 6-15. 最终的合并请求

如果你点开合并请求的“Files Changed”（更改的文件）选项卡，你将会看到“整理过的”差异表——也就是这个分支被合并到主分支之后将会产生的所有改动，其实就是 `git diff master...<分支名>` 命令的执行结果。你可以浏览 “确定引入了哪些东西” 来了解更多关于差异表的知识。

你还会注意到，GitHub 会检查你的合并请求是否能直接合并，如果可以，将会提供一个按钮来进行合并操作。这个按钮只在你对版本库有写入权限并且可以进行简洁合并时才会显示。你点击后 GitHub 将做出一个“非快进式”（non-fast-forward）合并，即使这个合并 能够 快进式（fast-forward）合并，

GitHub 依然会创建一个合并提交。

如果你需要，你还可以将分支拉取并在本地合并。如果你将这个分支合并到 `master` 分支中并推送到 GitHub，这个合并请求会被自动关闭。

这就是大部分 GitHub 项目使用的工作流程。创建分支，基于分支创建合并请求，进行讨论，根据需要继续在分支上进行修改，最终关闭或合并合并请求。

> 不必总是 Fork
>
> 有件很重要的事情：你可以在同一个版本库中不同的分支提交合并请求。如果你正在和某人实现某个功能，而且你对项目有写权限，你可以推送分支到版本库，并在 `master` 分支提交一个合并请求并在此进行代码审查和讨论的操作。不需要进行“Fork”。

合并请求的进阶用法

目前，我们学到了如何在 GitHub 平台对一个项目进行最基础的贡献。现在我们会教给你一些小技巧，让你可以更加有效率地使用合并请求。

将合并请求制作成补丁

有一件重要的事情：许多项目并不认为合并请求可以作为补丁，就和通过邮件列表工作的的项目对补丁贡献的看法一样。大多数的 GitHub 项目将合并请求的分支当作对改动的交流方式，并将变更集合起来统一进行合并。

这是个重要的差异，因为一般来说改动会在代码完成前提出，这和基于邮件列表的补丁贡献有着天差地别。这使得维护者们可以更早的沟通，由社区中的力量能提出更好的方案。当有人从合并请求提交了一些代码，并且维护者和社区提出了一些意见，这个补丁系列并不需要从头来过，只需要将改动重新提交并推送到分支中，这使得讨论的背景和过程可以齐头并进。

举个例子，你可以回去看看 Figure 6-15，你会注意到贡献者没有变基他的提交再提交一个新的合并请求，而是直接增加了新的提交并推送到已有的分支中。如果你之后再回去查看这个合并请求，你可以轻松地找到这个修改的原因。点击网页上的“Merge”（合并）按钮后，会建立一个合并提交并指向这个合并请求，你就可以很轻松的研究原来的讨论内容。

与上游保持同步

如果你的合并请求由于过时或其他原因不能干净地合并，你需要进行修复才能让维护者对其进行合并。GitHub 会对每个提交进行测试，让你知道你的合并请求能否简洁的合并。



Figure 6-16. 不能进行干净合并

如果你看到了像 Figure 6-16 中的画面，你就需要修复你的分支让这个提示变成绿色，这样维护者就不需要再做额外的工作。

你有两种方法来解决这个问题。你可以把你的分支变基到目标分支中去（通常是你派生出的版本库中的 `master` 分支），或者你可以合并目标分支到你的分支中去。

GitHub 上的大多数的开发者会使用后一种方法，基于我们在上一节提到的理由：我们最看重的是历史记录和最后的合并，变基除了给你带来看上去简洁的历史记录，只会让你的工作变得更加困难且更容易犯错。

如果你想要合并目标分支来让你的合并请求变得可合并，你需要将源版本库添加为一个新的远端，并从远端抓取内容，合并主分支的内容到你的分支中去，修复所有的问题并最终重新推送回你提交合并请求使用的分支。

在这个例子中，我们再次使用之前的“tonychacon”用户来进行示范，源作者提交了一个改动，使得合并请求和它产生了冲突。现在来看我们解决这个问题的步骤。

```
$ git remote add upstream https://github.com/schacon/blink

$ git fetch upstream
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
Unpacking objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
From https://github.com/schacon/blink
 * [new branch]      master     -> upstream/master

$ git merge upstream/master
Auto-merging blink.ino
CONFLICT (content): Merge conflict in blink.ino
Automatic merge failed; fix conflicts and then commit the result.

$ vim blink.ino
$ git add blink.ino
$ git commit
[slow-blink 3c8d735] Merge remote-tracking branch 'upstream/master' \
    into slower-blink

$ git push origin slow-blink
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 682 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
To https://github.com/tonychacon/blink
   ef4725c..3c8d735  slower-blink -> slow-blink
```

❶ 将源版本库添加为一个远端，并命名为“upstream”（上游）

❷ 从远端抓取最新的内容

❸ 将主分支的内容合并到你的分支中

❹ 修复产生的冲突

❺ 再推送回同一个分支

你完成了上面的步骤后，合并请求将会自动更新并重新检查是否能干净的合并。

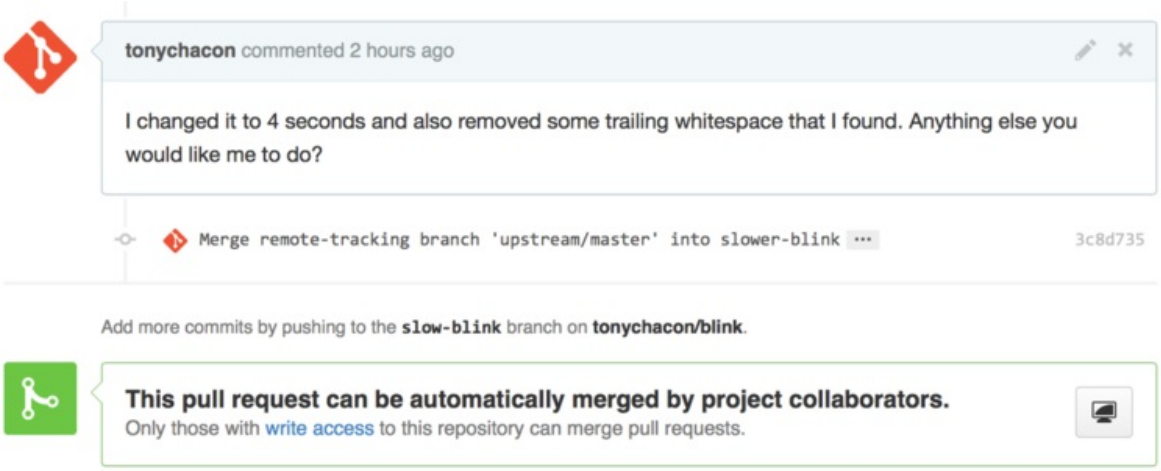


Figure 6-17. 合并请求现在可以干净地合并了

Git 的伟大之处就是你可以一直重复以上操作。如果你有一个运行了十分久的项目，你可以轻松地合并目标分支且只需要处理最近的一次冲突，这使得管理流程更加容易。

如果你一定想对分支做变基并进行清理，你可以这么做，但是强烈建议你不要强行的提交到已经提交了合并请求的分支。如果其他人拉取了这个分支并进行一些修改，你将会遇到 “变基的风险” 中提到的问题。相对的，将变基后的分支推送到 GitHub 上的一个新分支中，并且创建一个全新的合并请求引用旧的合并请求，然后关闭旧的合并请求。

参考
你的下个问题可能是“我该如何引用旧的合并请求？”。有许多方法可以让你在 GitHub 上的几乎任何地方引用其他东西。

先从如何对合并请求或议题（Issue）进行相互引用开始。所有的合并请求和议题在项目中都会有一个独一无二的编号。举个例子，你无法同时拥有 3 号合并请求和 3 号议题。如果你想要引用任何一个合并请求或议题，你只需要在提交或描述中输入 `#<编号>` 即可。你也可以指定引用其他版本库的议题或合并请求，如果你想要引用其他人对该版本库的“Fork”中的议题或合并请求，输入 `用户名#<编号>` ，如果在不同的版本库中，输入 `用户名/版本库名#<编号>` 。

我们来看一个例子。假设我们对上个例子中的分支进行了变基，并为此创建一个新的合并请求，现在我们希望能在新的合并请求中引用旧的合并请求。我们同时希望引用一个派生出的项目中的议题和一个完全不同的项目中的议题，就可以像 Figure 6-18 这样填写描述。

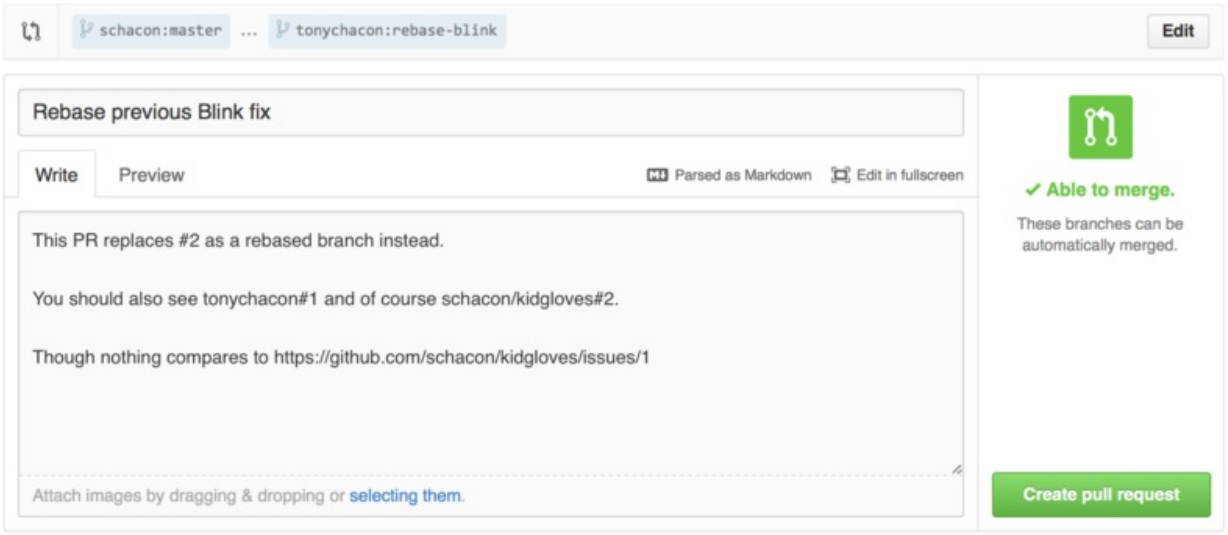


Figure 6-18. 在合并请求中的交叉引用

当我们提交了这个合并请求，我们将会看到以上内容被渲染成这样：Figure 6-19

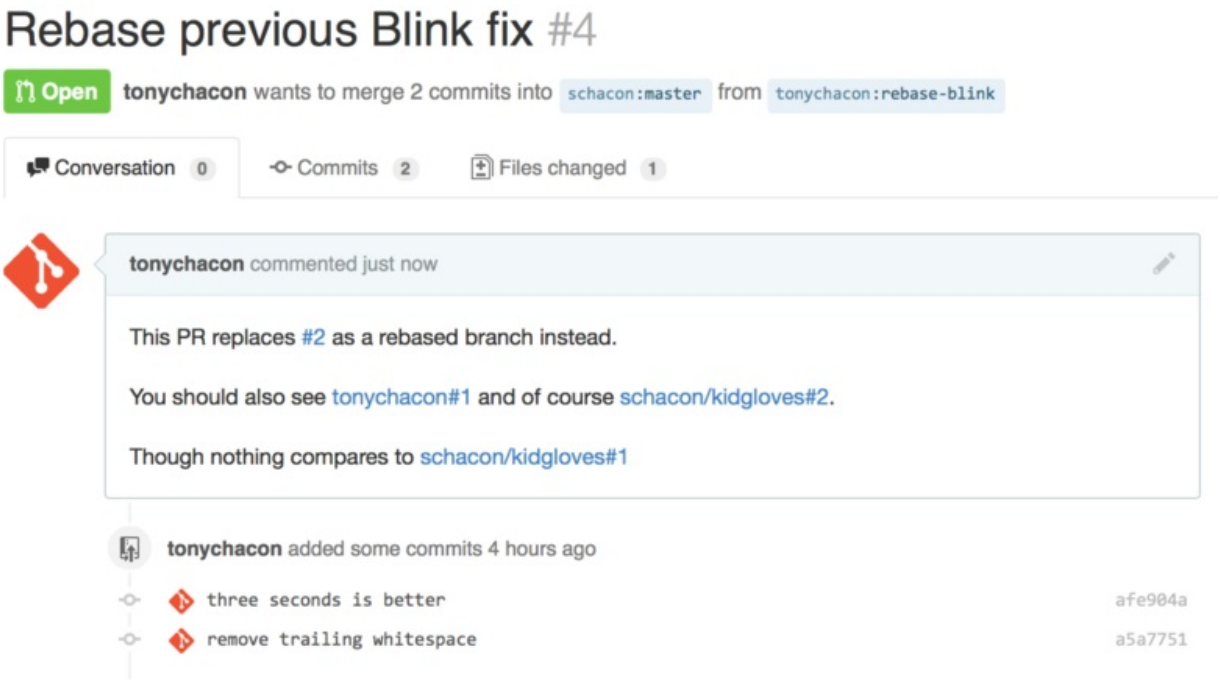


Figure 6-19. 在合并请求中渲染后的交叉引用

你会注意到完整的 GitHub 地址被简化了，只留下了必要的信息。

如果 Tony 回去关闭了源合并请求，我们可以看到一个被引用的提示，GitHub 会自动的反向追踪事件并显示在合并请求的时间轴上。这意味着任何查看这个合并请求的人可以轻松地访问新的合并请求。这个链接就像 Figure 6-20 中展示的那样。

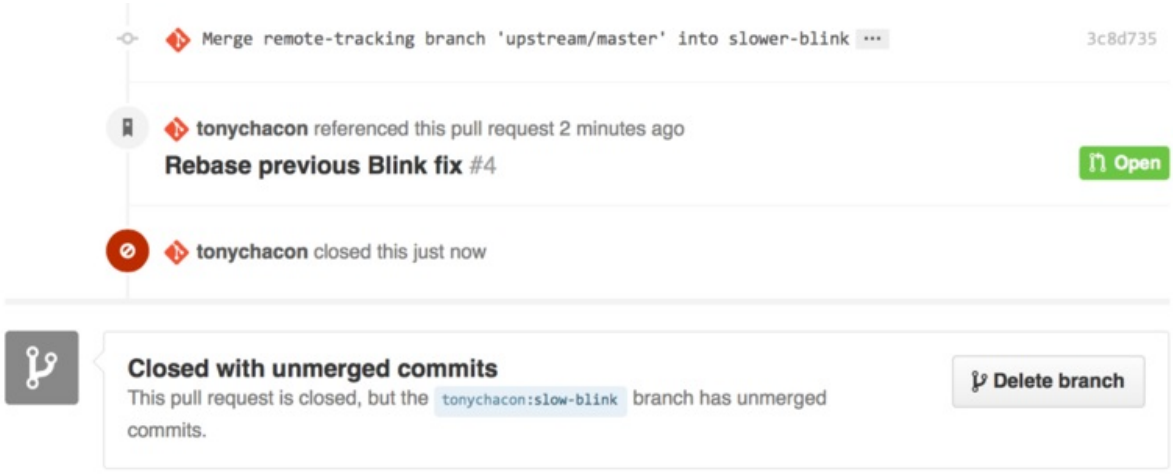


Figure 6-20. 在合并请求中渲染后的交叉引用

除了议题编号外，你还可以通过使用提交的 SHA-1 来引用提交。你必须完整的写出 40 位长的 SHA，GitHub 会在评论中自动地产生指向这个提交的链接。同样的，你可以像引用议题一样对“Fork”出的项目中的提交或者其他项目中的提交进行引用。

Markdown

对于在 GitHub 中绝大多数文本框中能够做到的事，引用其他议题只是个开始。在议题和合并请求的描述，评论和代码评论还有其他地方，都可以使用“GitHub 风格的 Markdown”。Markdown 可以让你输入纯文本，但是渲染出丰富的内容。

查看 Figure 6-21 里的例子来了解如何书写评论或文本，并通过 Markdown 进行渲染。

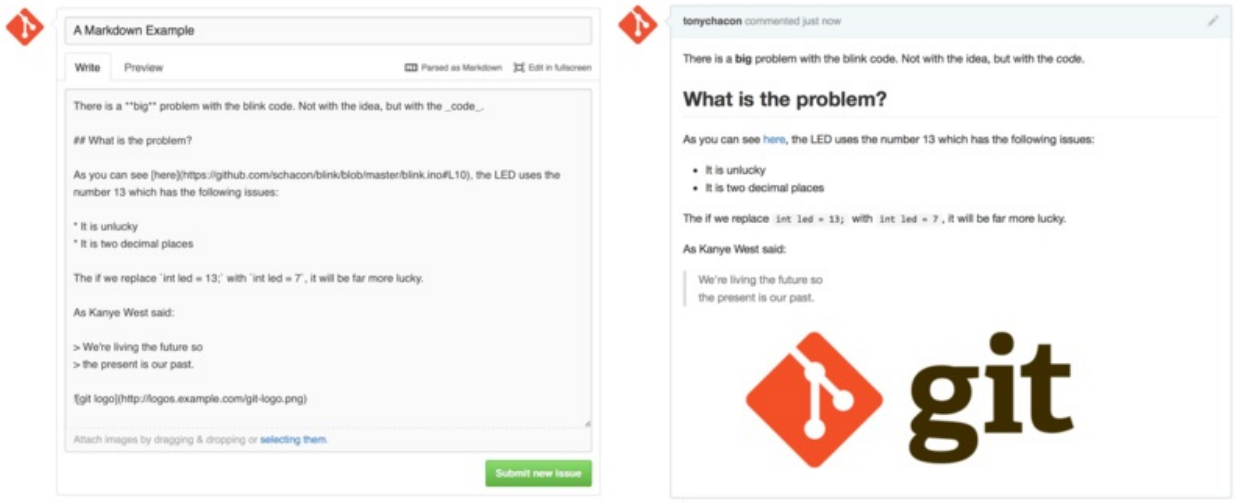


Figure 6-21. 一个 Markdown 的例子和渲染效果

GitHub 风格的 Markdown

GitHub 风格的 Markdown 增加了一些基础的 Markdown 中做不到的东西。它在创建合并请求和议题中的评论和描述时十分有用。

任务列表
第一个 GitHub 专属的 Markdown 功能，特别是用在合并请求中，就是任务列表。一个任务列表可以展示出一系列你想要完成的事情，并带有复选框。把它们放在议题或合并请求中时，通常可以展示你想要完成的事情。

你可以这样创建一个任务列表：

```
- [X] 编写代码
- [ ] 编写所有测试程序
- [ ] 为代码编写文档
```

如果我们将这个列表加入合并请求或议题的描述中，它将会被渲染 Figure 6-22 这样。



Figure 6-22. Markdown 评论中渲染后的任务列表

在合并请求中，任务列表经常被用来在合并之前展示这个分支将要完成的事情。最酷的地方就是，你只需要点击复选框，就能更新评论 —— 你不需要直接修改 Markdown。

不仅如此，GitHub 还会将你在议题和合并请求中的任务列表整理起来集中展示。举个例子，如果你在一个合并请求中有任务清单，你将会在所有合并请求的总览页面上看到它的进度。这使得人们可以把一个合并请求分解成不同的小任务，同时便于其他人了解分支的进度。你可以在 Figure 6-23 看到一个例子。



Figure 6-23. 在合并请求列表中的任务列表总结

当你在实现一个任务的早期就提交合并请求，并使用任务清单追踪你的进度，这个功能会十分的有用。

摘录代码

你也可以在评论中摘录代码。这在你想要展示尚未提交到分支中的代码时会十分有用。它也经常被用在展示无法正常工作的代码或这个合并请求需要的代码。

你需要用“反引号”将需要添加的摘录代码包起来。

```
```java
for(int i=0 ; i < 5 ; i++)
{
 System.out.println("i is : " + i);
}
```
```

如果加入语言的名称，就像我们这里加入的“java”一样，GitHub 会自动尝试对摘录的片段进行语法高亮。在下面的例子中，它最终会渲染成这个样子： Figure 6-24 。



Figure 6-24. 渲染后的摘录代码示例

引用

如果你在回复一个很长的评论之中的一小段，你只需要复制你需要的片段，并在每行前添加 `>` 符号即可。事实上，因为这个功能会被经常用到，它也有一个快捷键。只要你把你要回应的文字选中，并按下 `r` 键，选中的问题会自动引用并填入评论框。

引用的部分就像这样:

```
> Whether 'tis Nobler in the mind to suffer
> The Slings and Arrows of outrageous Fortune,

How big are these slings and in particular, these arrows?
```

经过渲染后，就会变成这样： Figure 6-25

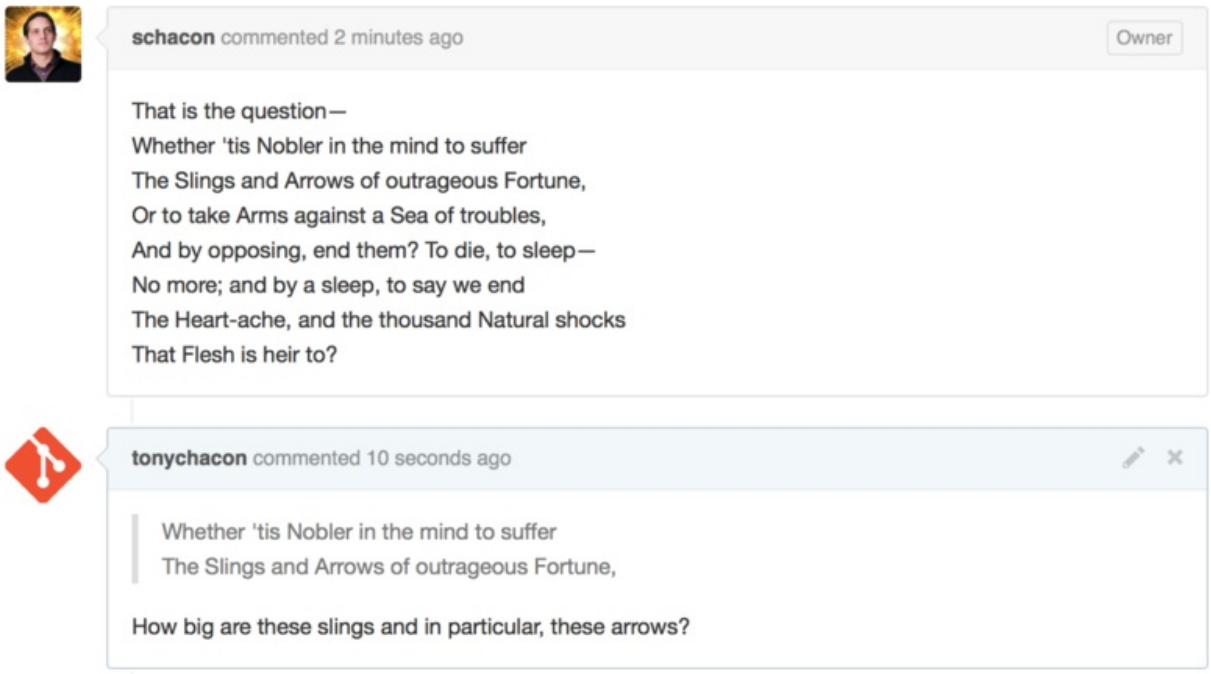


Figure 6-25. 渲染后的引用示例

表情符号（Emoji）
最后，我们可以在评论中使用表情符号。这经常出现在 GitHub 的议题和合并请求的评论中。GitHub 上甚至有表情助手。如果你在输入评论时以 `:` 开头，自动完成器会帮助你找到你需要的表情。

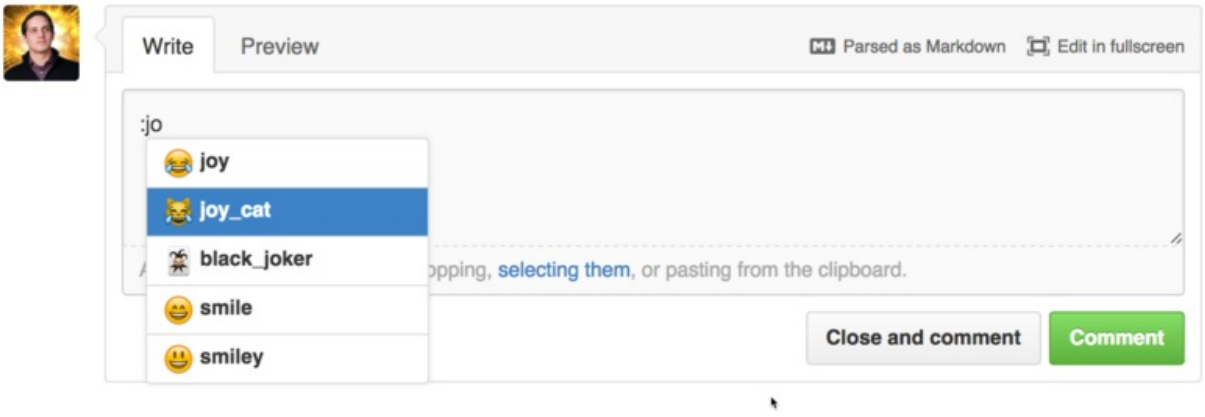


Figure 6-26. 表情符号自动完成器

你也可以在评论的任何地方使用 `:<表情名称>:` 来添加表情符号。举个例子，你可以输入以下文字：

```
I :eyes: that :bug: and I :cold_sweat:.

:trophy: for :microscope: it.
```

```
:+1: and :sparkles: on this :ship:, it's :fire::poop:!

:clap::tada::panda_face:
```

渲染之后，就会变成这样： Figure 6-27

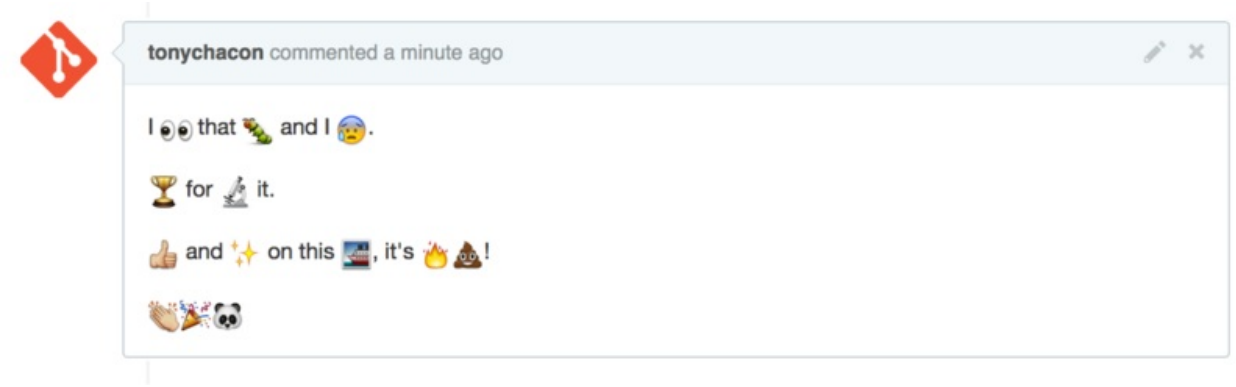


Figure 6-27. 使用了大量表情符号的评论

虽然这个功能并不是非常实用，但是它在这种不方便表达感情的媒体里，加入了趣味的元素。

事实上现在已经有大量的在线服务可以使用表情符号，这里有个列表可以让你快速的找到能表达你的情绪的表情符号：

http://www.emoji-cheat-sheet.com

图片
从技术层面来说，这并不是 GitHub 风格 Markdown 的功能，但是也很有用。如果不想使用 Markdown 语法来插入图片，GitHub 允许你通过拖拽图片到文本区来插入图片。

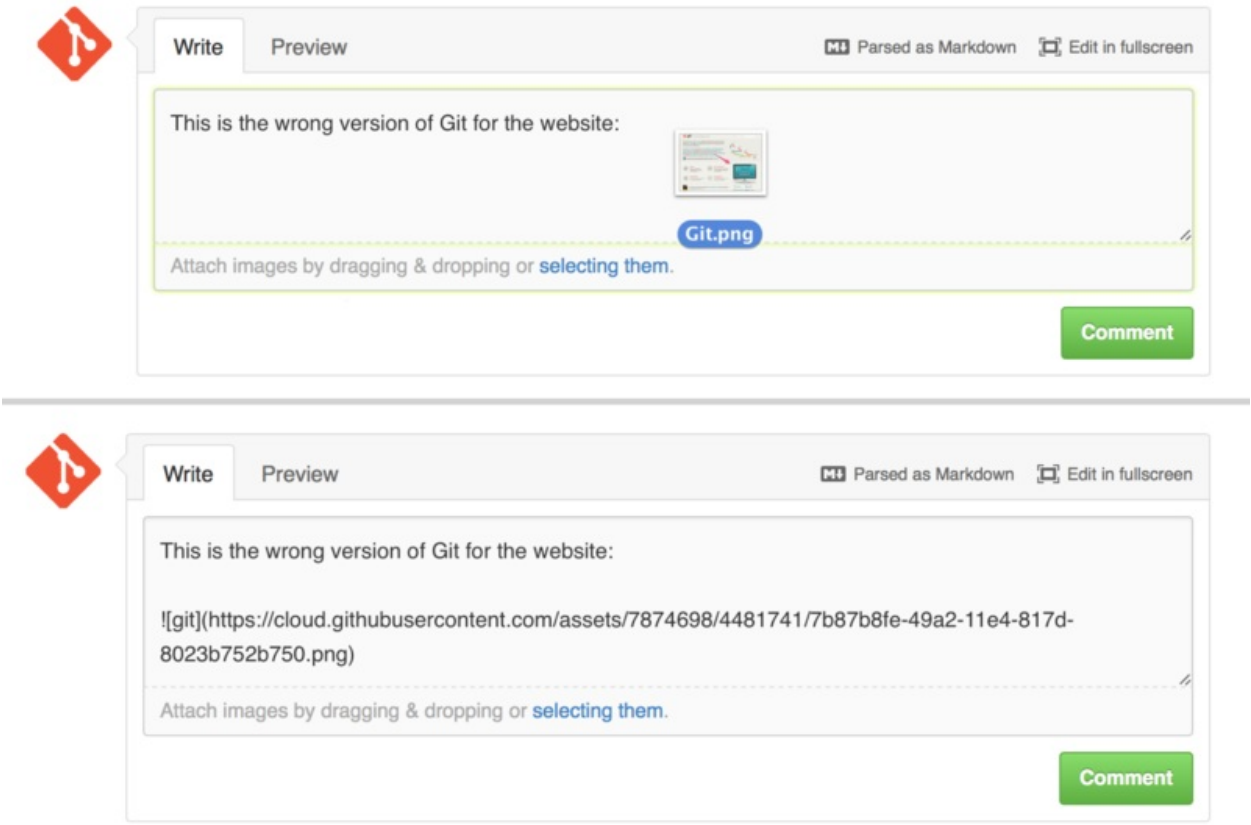


Figure 6-28. 通过拖拽的方式自动插入图片

如果你回去查看 Figure 6-18 ，你会发现文本区上有个“Parsed as Markdown”的提示。点击它你可以了解所有能在 GitHub 上使用的 Markdown 功能。

6.3 维护项目

维护项目

现在我们可以很方便地向一个项目贡献内容，来看一下另一个方面的内容：创建、维护和管理你自己的项目。

创建新的版本库

让我们创建一个版本库来分享我们的项目。通过点击面板右侧的“New repository”按钮，或者顶部工具条你用户名旁边的 `+` 按钮来开始我们的旅程。 参见 Figure 6-30。



Figure 6-29. 这是 “Your repositories” 区域.

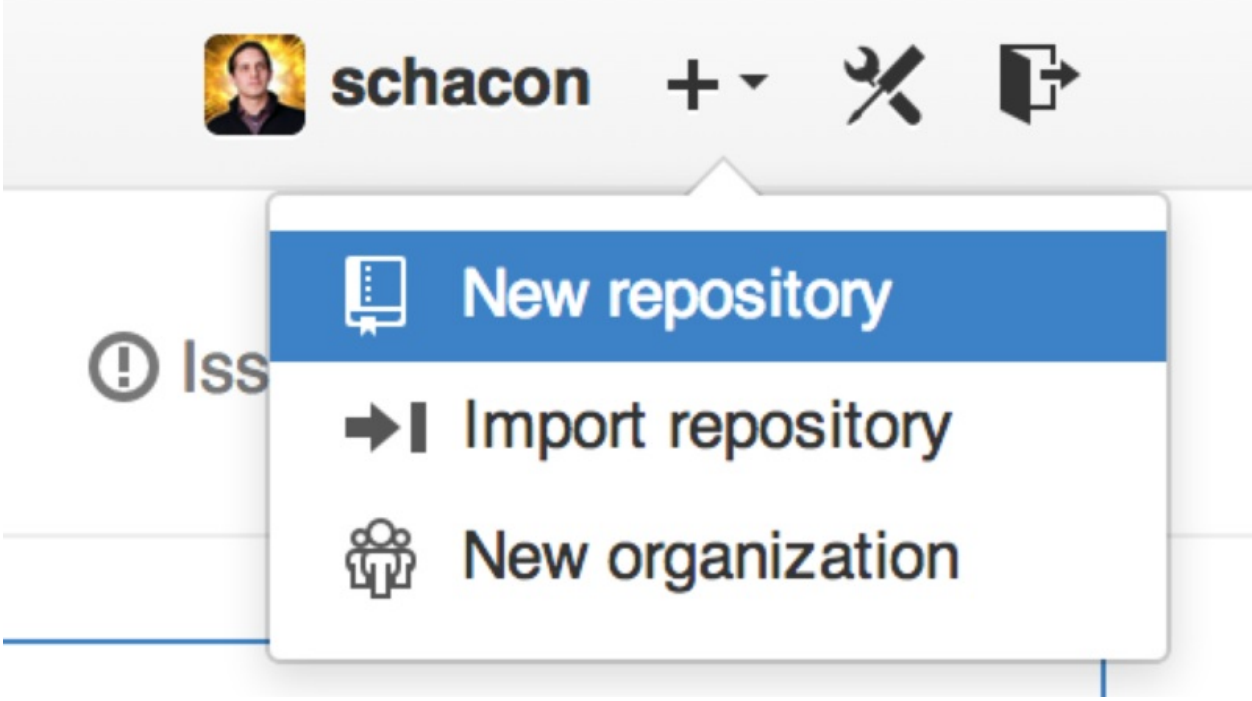


Figure 6-30. 这是 “New repository” 下拉列表.

这会带你到 “new repository” 表单:

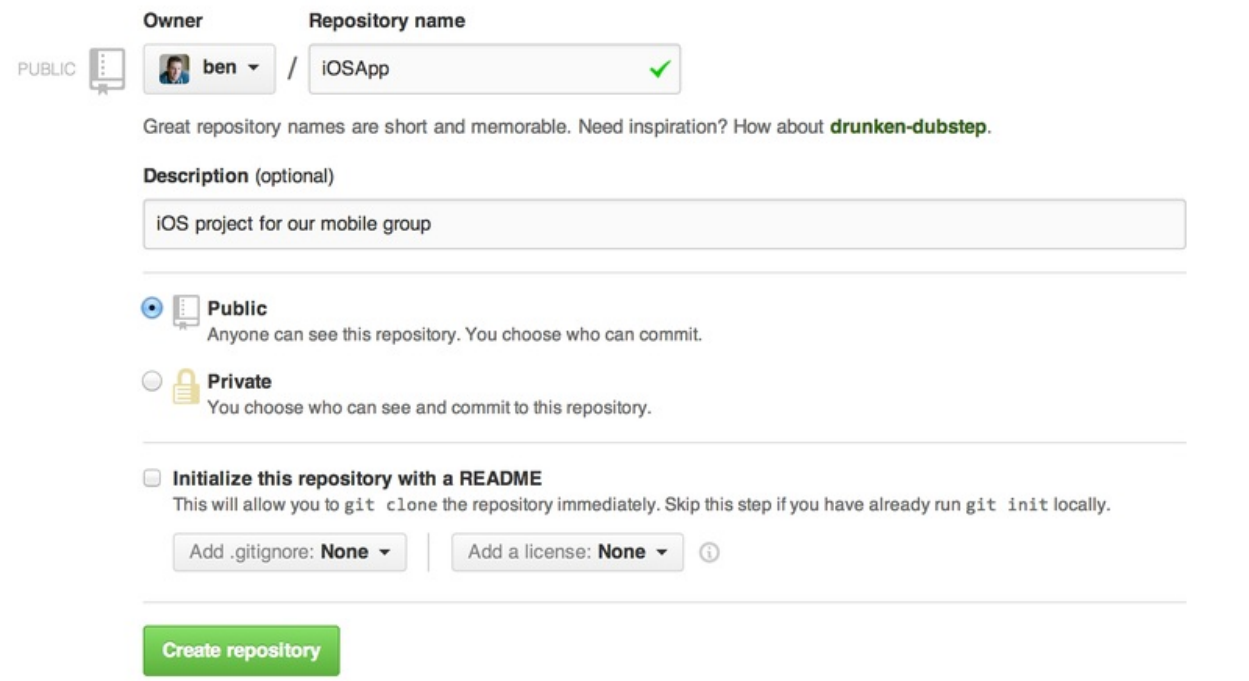


Figure 6-31. 这是 “new repository” 表单.

这里除了一个你必须要填的项目名，其他字段都是可选的。现在只需要点击 “Create Repository” 按

钮，Duang!!! – 你就在 GitHub 上拥有了一个以 `<user>/<project_name>` 命名的新仓库了。

因为目前暂无代码，GitHub 会显示有关创建新版本库或者关联到一个已有的 Git 版本库的一些说明。我们不会在这里详细说明此项，如果你需要复习，去看 Chapter 2。

现在你的项目就托管在 GitHub 上了，你可以把 URL 给任何你想分享的人。GitHub 上的项目可通过 HTTP 或 SSH 访问，格式是：HTTP ： `https://github.com/<user>/<project_name>` ，SSH ： `git@github.com:<user>/<project_name>` 。Git 可以通过以上两种 URL 进行抓取和推送，但是用户的访问权限又因连接时使用的证书不同而异。

通常对于公开项目可以优先分享基于 HTTP 的 URL，因为用户克隆项目不需要有一个 GitHub 帐号。如果你分享 SSH URL，用户必须有一个帐号并且上传 SSH 密钥才能访问你的项目。HTTP URL 与你贴到浏览器里查看项目用的地址是一样的。

添加合作者

如果你想与他人合作，并想给他们提交的权限，你需要把他们添加为 “Collaborators” 。如果 Ben，Jeff，Louise 都在 GitHub 上注册了，你想给他们推送的权限，你可以将他们添加到你的项目。这样做会给他们 “推送” 权限，就是说他们对项目和 Git 版本库都有读写的权限。

点击边栏底部的 “Settings” 链接。

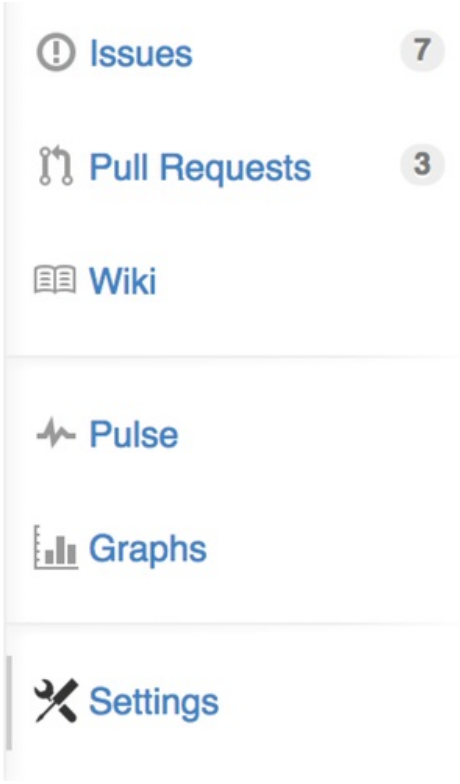


Figure 6-32. 版本库设置链接.

然后从左侧菜单中选择 “Collaborators” 。然后，在输入框中填写用户名，点击 “Add collaborator.” 如果你想授权给多个人，你可以多次重复这个步骤。如果你想收回权限，点击他们同一行右侧的 “X”

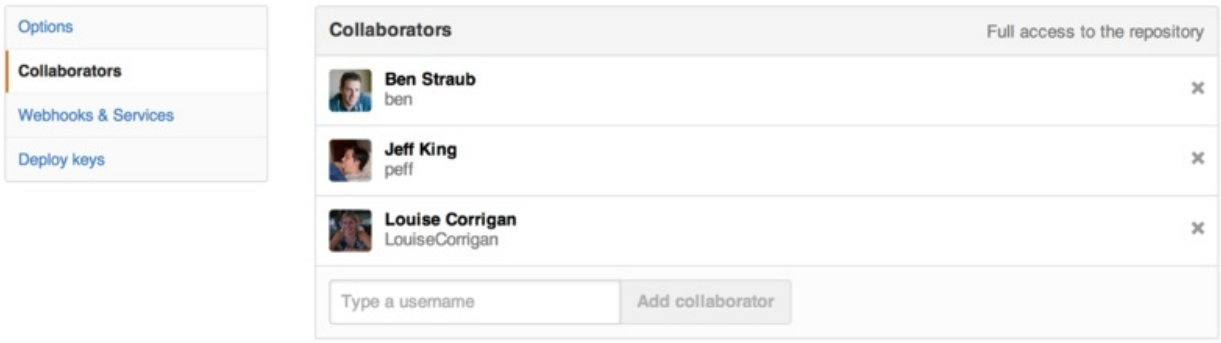


Figure 6-33. 版本库合作者.

管理合并请求

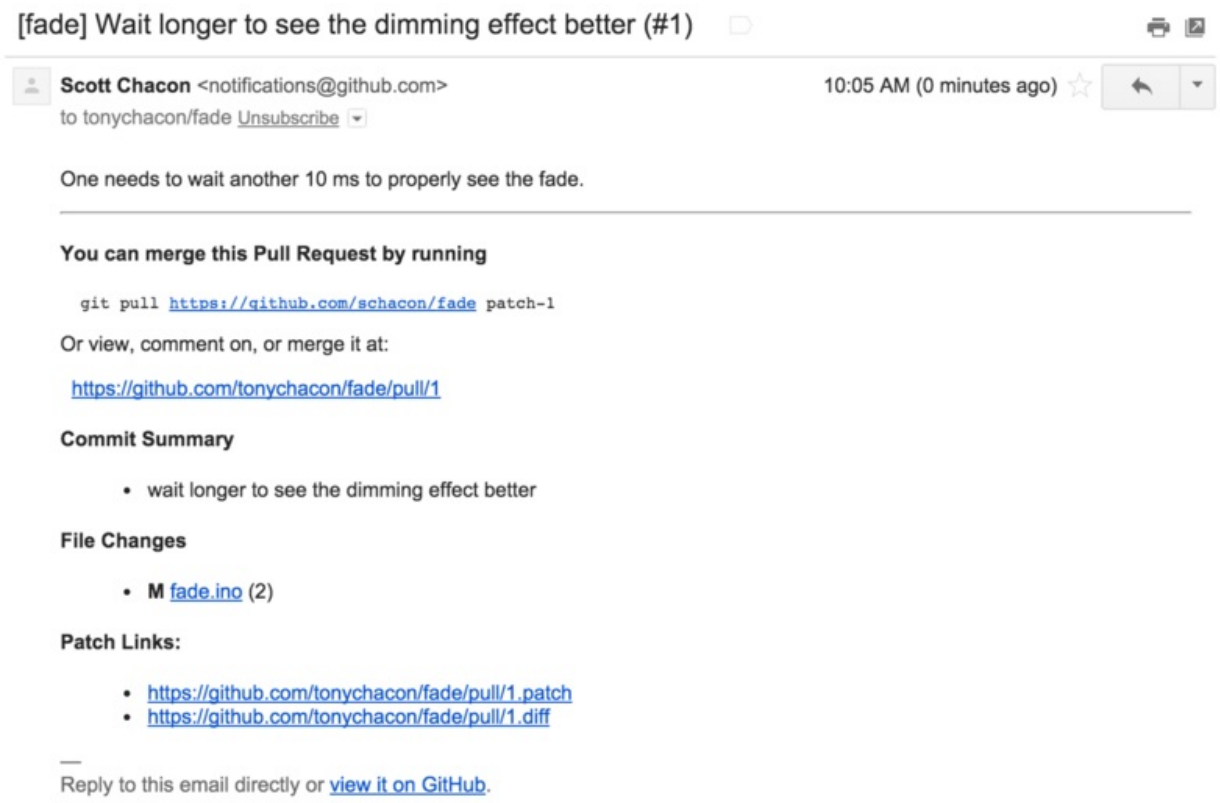
现在你有一个包含一些代码的项目，可能还有几个有推送权限的合作者，下面来看当你收到合并请求时该做什么。

合并请求可以来自仓库副本的一个分支，或者同一仓库的另一个分支。唯一的区别是 fork 过来的通常是和你不能互相推送的人，而内部的推送通常都可以互相访问。

作为例子，假设你是 “tonychacon” ，你创建了一个名为 “fade” 的 Arduino 项目.

邮件通知

有人来修改了你的代码，给你发了一个合并请求。你会收一封关于合并请求的提醒邮件，它看起来像 Figure 6-34。


[fade] Wait longer to see the dimming effect better (#1)

Scott Chacon <notifications@github.com> 10:05 AM (0 minutes ago)
to tonychacon/fade Unsubscribe

One needs to wait another 10 ms to properly see the fade.

You can merge this Pull Request by running

```
git pull https://github.com/schacon/fade patch-1
```

Or view, comment on, or merge it at:

https://github.com/tonychacon/fade/pull/1

Commit Summary

- wait longer to see the dimming effect better

File Changes

- **M** fade.ino (2)

Patch Links:

- https://github.com/tonychacon/fade/pull/1.patch
- https://github.com/tonychacon/fade/pull/1.diff

—
Reply to this email directly or view it on GitHub.

Figure 6-34. 新的合并请求的邮件通知.

关于这个邮件有几个要注意的地方。它会给你一个小的变动统计结果 — 一个包含合并请求中改变的文件和改变了多少的列表。它还给你一个 GitHub 上进行合并请求操作的链接。还有几个可以在命令行使用的 URL。

如果你注意到 `git pull <url> patch-1` 这一行，这是一种合并远程分支的简单方式，无需必须添加一个远程分支。我们很快会在 “检出远程分支”._ 讲到它。如果你愿意，你可以创建并切换到一个主题分支，然后运行这个命令把合并请求合并进来。

还有一些有趣的 URL，像 `.diff` 和 `.patch` ，就像你猜的那样，它们提供 diff 和 patch 的标准版本。你可以技术性地用下面的方法合并“合并请求” ：

```
$ curl http://github.com/tonychacon/fade/pull/1.patch | git am
```

在合并请求上进行合作

就像我们在 “GitHub 流程” ，_ 说过的，现在你可以跟开启合并请求的人进行会话。你既可以对某些代码添加注释，也可以对整个提交添加注释或对整个合并请求添加注释，在任何地方都可以用 GitHub Flavored Markdown。

每次有人在合并请求上进行注释你都会收到通知邮件，通知你哪里发生改变。他们都会包含一个到改变位置的链接，你可以直接在邮件中对合并请求进行注释。

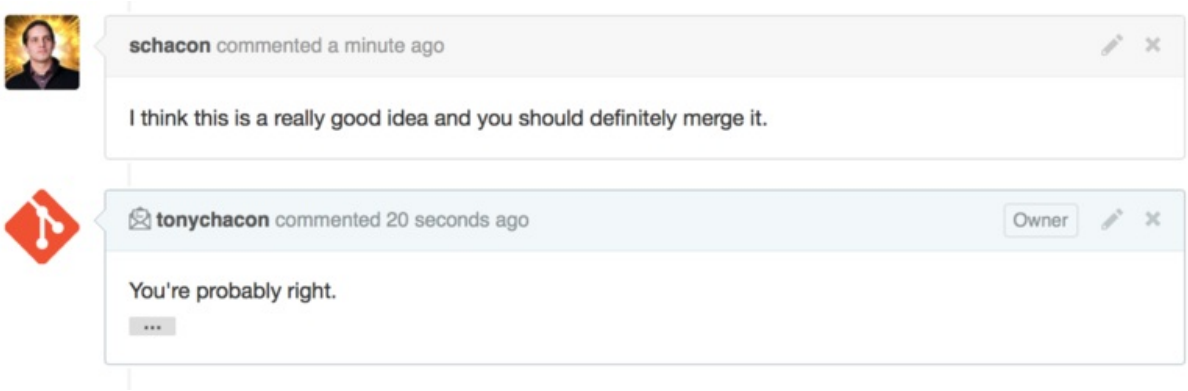


Figure 6-35. Responses to emails are included in the thread.

一旦代码符合了你的要求，你想把它合并进来，你可以把代码拉取下来在本地进行合并，也可以用我们之前提到过的 `git pull <url> <branch>` 语法，或者把 fork 添加为一个 remote，然后进行抓取和合并。

对于很琐碎的合并，你也可以用 GitHub 网站上的 “Merge” 按钮。它会做一个 “non-fast-forward” 合并，即使可以快进（fast-forward）合并也会产生一个合并提交记录。就是说无论如何，只要你点击 merge 按钮，就会产生一个合并提交记录。你可以在 Figure 6-36 看到，如果你点击提示链接，GitHub 会给你所有的这些信息。

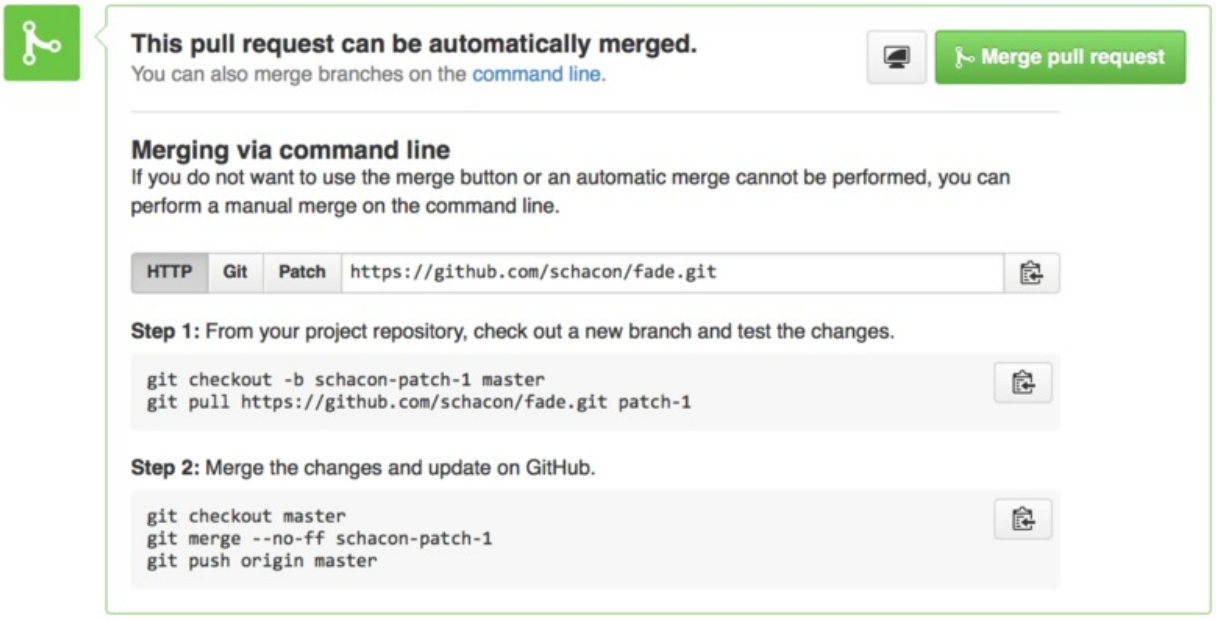


Figure 6-36. 合并按钮和手工合并一个合并请求的指令.

如果你决定不合并它，你可以把合并请求关掉，开启合并请求的人会收到通知。

合并请求引用

如果你正在处理 许多 合并请求，不想添加一堆 remote 或者每次都要做一次拉取，这里有一个可以在 GitHub 上用的小技巧。这是有点高级的技巧，但它相当有用，我们会在 “引用规格” 有更多的细节说明。

实际上 GitHub 在服务器上把合并请求分支视为一种 “假分支”。默认情况下你克隆时不会得到它们，但它们还是隐式地存在，你可以很容易地访问到它们。

为了展示这个，我们要用到一个叫做 `ls-remote` 的低级命令（通常被叫做“plumbing”，我们会在 “底层命令和高层命令” 读到更多相关内容）。这个命令在日常 Git 操作中基本不会用到，但在显示服务器上有哪些引用（reference）时很管用。

如果在我们之前用过的 “blink” 版本库上使用这个命令，我们会得到一个版本库里所有的分支，标签和其它引用（reference）的列表。

```
$ git ls-remote https://github.com/schacon/blink
10d539600d86723087810ec636870a504f4fee4d        HEAD
10d539600d86723087810ec636870a504f4fee4d        refs/heads/master
6a83107c62950be9453aac297bb0193fd743cd6e        refs/pull/1/head
afe83c2d1a70674c9505cc1d8b7d380d5e076ed3        refs/pull/1/merge
3c8d735ee16296c242be7a9742ebfbc2665adec1        refs/pull/2/head
15c9f4f80973a2758462ab2066b6ad9fe8dcf03d        refs/pull/2/merge
a5a7751a33b7e86c5e9bb07b26001bb17d775d1a        refs/pull/4/head
31a45fc257e8433c8d8804e3e848cf61c9d3166c        refs/pull/4/merge
```

当然，如果你在你自己的版本库或其它你想检查的远程版本库中使用 `git ls-remote origin` ，它会显示相似的内容。

如果版本库在 GitHub 上并且有打开的合并请求，你会得到一些以 `refs/pull/` 开头的引用。它们实际上是分支，但因为它们不在 `refs/heads/` 中，所以正常情况下你克隆时不会从服务器上得到它们 — 抓取过程正常情况下会忽略它们。

每个合并请求有两个引用 - 其中以 `/head` 结尾的引用指向的提交记录与合并请求分支中的最后一个提交记录是同一个。所以如果有人在我们的版本库中开启了一个合并请求，他们的分支叫做 `bug-fix` ，指向 `a5a775` 这个提交记录，那么在 我们的 版本库中我们没有 `bug-fix` 分支（因为那是在他们的 fork 中），但我们 可以 有一个 `pull/<pr#>/head` 指向 `a5a775` 。这意味着我们可以很容易地拉取每一个合并请求分支而不用添加一堆 remote。

现在，你可以像直接抓取引用一样抓取那些分支或提交。

```
$ git fetch origin refs/pull/958/head
From https://github.com/libgit2/libgit2
 * branch            refs/pull/958/head -> FETCH_HEAD
```

这告诉 Git： “连接到 `origin` 这个 remote，下载名字为 `refs/pull/958/head` 的引用。” Git 高高兴兴去执行，下载构建那个引用需要的所有内容，然后把指针指向 `.git/FETCH_HEAD` 下面你想要的提交记录。然后你可以用 `git merge FETCH_HEAD` 把它合并到你想进行测试的分支，但那个合并的提交信息看起来有点怪。然而，如果你需要审查 一大批 合并请求，这样操作会很麻烦。

还有一种方法可以抓取 所有的 合并请求，并且在你连接到远程（remote）的时候保持更新。用你最喜欢的编辑器打开 `.git/config` ，查找 `origin` 远程（remote）。看起来差不多像下面这样：

```
[remote "origin"]
    url = https://github.com/libgit2/libgit2
    fetch = +refs/heads/*:refs/remotes/origin/*
```

以 `fetch =` 开头的行是一个 “refspec.” 它是一种把 remote 的名称映射到你本地 `.git` 目录的方法。这一条（就是上面的这一条）告诉 Git， “remote 上 `refs/heads` 下面的内容在我本地版本库中都放在 `refs/remotes/origin` 。” 你可以把这一段修改一下，添加另一个 refspec：

```
[remote "origin"]
    url = https://github.com/libgit2/libgit2.git
    fetch = +refs/heads/*:refs/remotes/origin/*
    fetch = +refs/pull/*/head:refs/remotes/origin/pr/*
```

最后一行告诉 Git： “所有看起来像 `refs/pull/123/head` 的引用应该在本地版本库像 `refs/remotes/origin/pr/123` 一样存储” 现在，如果你保存那个文件，执行 `git fetch` ：

```
$ git fetch
# …
 * [new ref]         refs/pull/1/head -> origin/pr/1
 * [new ref]         refs/pull/2/head -> origin/pr/2
 * [new ref]         refs/pull/4/head -> origin/pr/4
# …
```

现在所有的合并请求在本地像分支一样展现，它们是只读的，当你执行抓取时它们也会更新。这让在本地测试合并请求中的代码变得超级简单：

```
$ git checkout pr/2
Checking out files: 100% (3769/3769),  done.
Branch pr/2 set up to track remote branch pr/2 from origin.
Switched to a new branch 'pr/2'
```

你的鹰眼系统会发现在 refspec 的 remote 部分的结尾有个 `head` 。在 GitHub 那边也有一个 `refs/pull/#/merge` 引用，它代表的是如果你在网站上按了 “merge” 按钮对应的提交记录。这甚至让你可以在按按钮之前就测试这个合并。

合并请求之上的合并请求

你不仅可以在主分支或者说 `master` 分支上开启合并请求，实际上你可以在网络上的任何一个分支上开启合并请求。其实，你甚至可以在另一个合并请求上开启一个合并请求。

如果你看到一个合并请求在向正确的方向发展，然后你想在这个合并请求上做一些修改或者你不太确定这是个好主意，或者你没有目标分支的推送权限，你可以直接在合并请求上开启一个合并请求。

当你开启一个合并请求时，在页面的顶端有一个框框显示你要合并到哪个分支和你从哪个分支合并过来的。如果你点击那个框框右边的 “Edit” 按钮，你不仅可以改变分支，还可以选择哪个 fork。

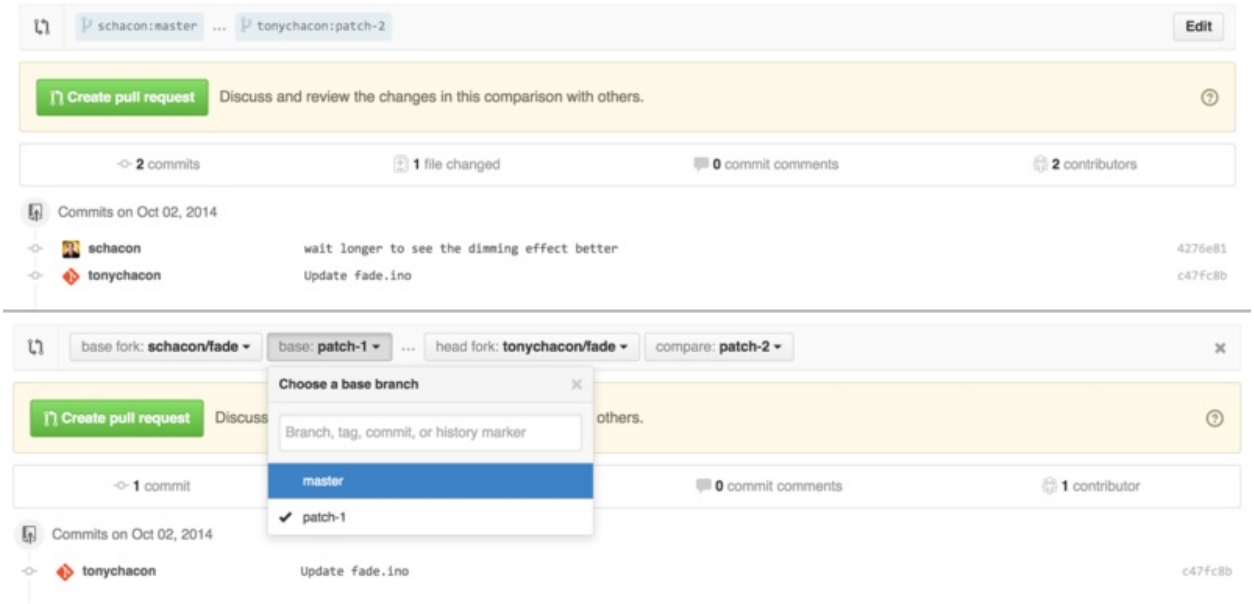


Figure 6-37. 手工修改合并请求的目标.

这里你可以很简单地指明合并你的分支到哪一个合并请求或 fork。

提醒和通知

GitHub 内置了一个很好的通知系统，当你需要与别人或别的团队交流时用起来很方便。

在任何评论中你可以先输入一个 `@`，系统会自动补全项目中合作者或贡献者的名字和用户名。

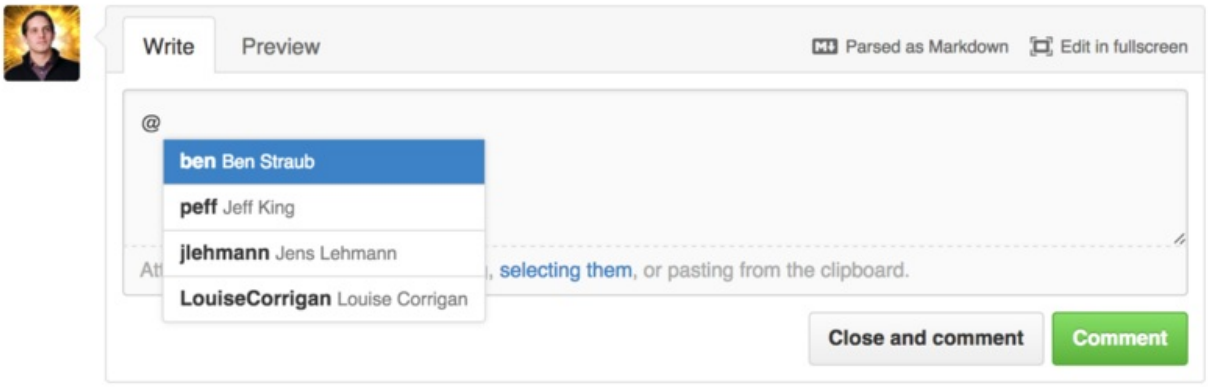


Figure 6-38. 输入 @ 来提醒某人.

你也可以提醒不在列表中的用户，但是通常自动补全用起更快。

当你发布了一个带用户提醒的评论，那个用户会收到通知。这意味着把人们拉进会话中要比让他们投票有效率得多。对于 GitHub 上的合并请求，人们经常把他们团队或公司中的其它人拉来审查问题或合并请求。

如果有人收到了合并请求或问题的提醒，他们会＂订阅＂它，后面有新的活动发生他们都会持续收到提醒。如果你是合并请求或者问题的发起方你也会被订阅上，比如你在关注一个版本库或者你评论了什么东西。如果你不想再收到提醒，在页面上有个 “Unsubscribe” 按钮，点一下就不会再收到更新了。

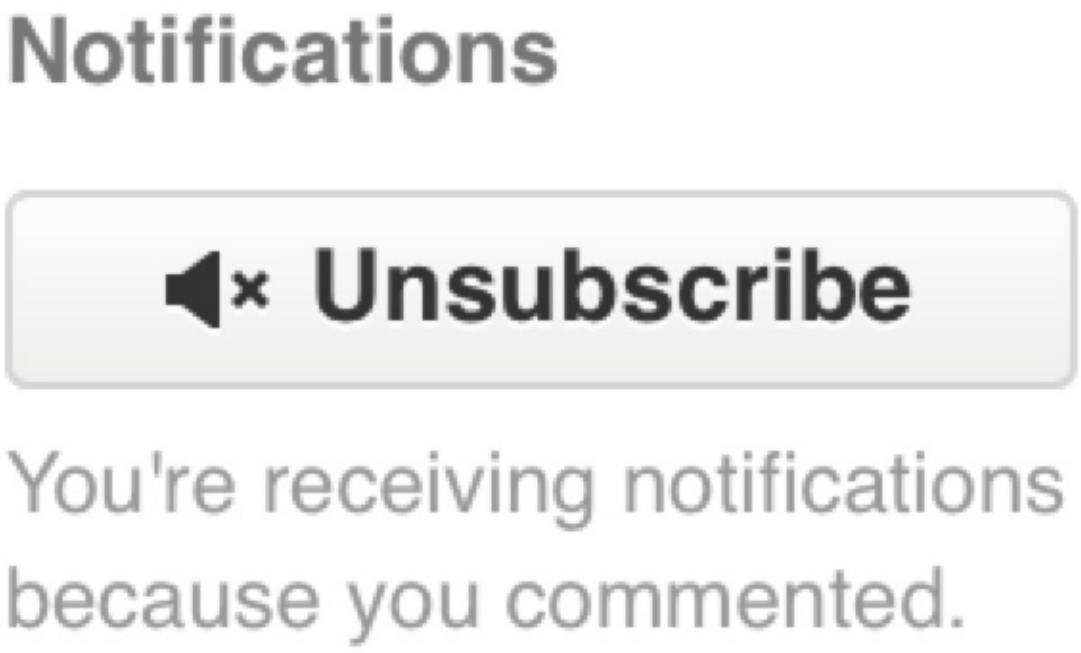


Figure 6-39. 取消订阅一个问题或合并请求.

通知页面

当我们在这提到特指 GitHub 的 “notifications” ，指的是当 GitHub 上有事件发生时，它通知你的方式，这里有几种不同的方式来配置它们。如果你打开配置页面的 “Notification center” 标签，你可以看到一些选项。

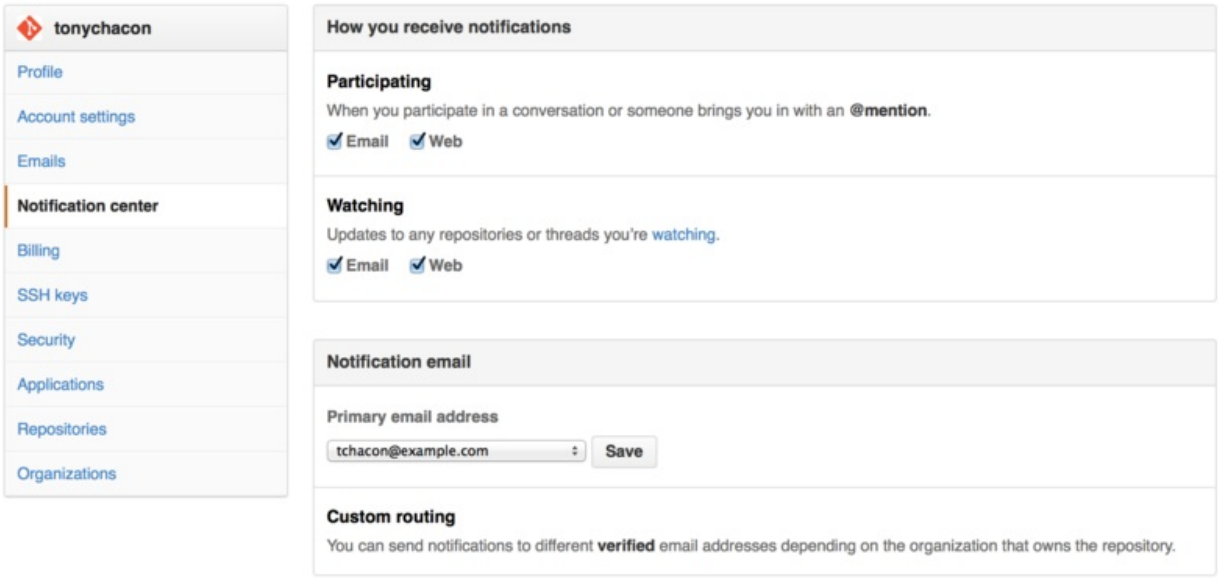


Figure 6-40. 通知中心选项.

有两个选项，通过 " 邮件(Email) " 和通过 " 网页(Web) " ，你可以选用一个或者都不选或者都选。

网页通知

网页通知只在 GitHub 上存在，你也只能在 GitHub 上查看。如果你打开了这个选项并且有一个你的通知，你会在你屏幕上方的通知图标上看到一个小蓝点。参见 Figure 6-41。

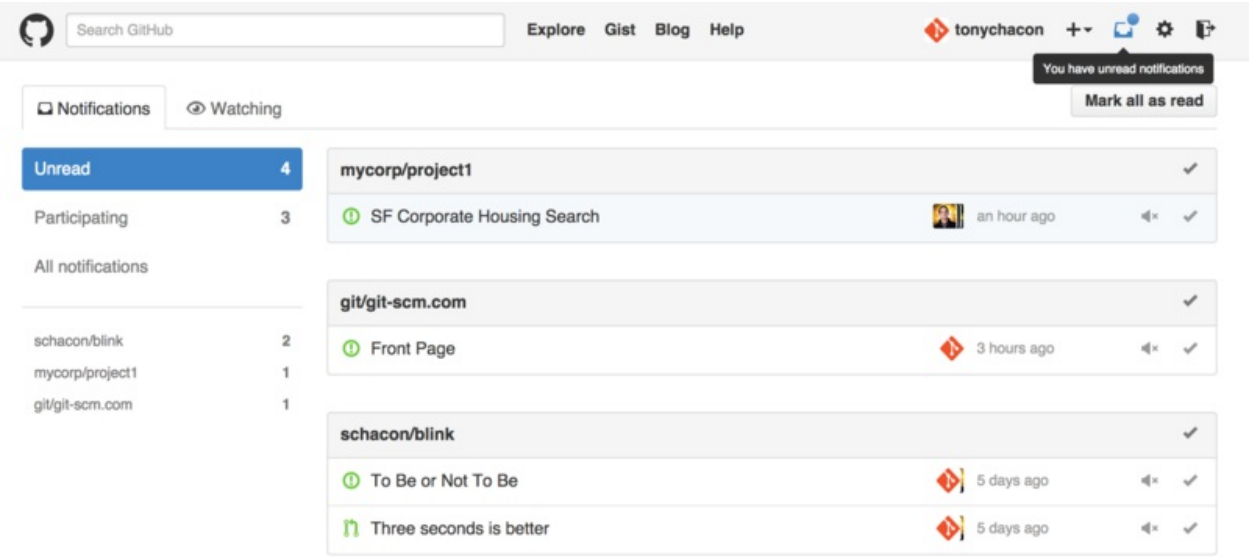


Figure 6-41. 通知中心.

如果你点击那个玩意儿，你会看到你被通知到的所有条目，按照项目分好了组。你可以点击左边栏的项目

名字来过滤项目相关的通知。你可以点击通知旁边的对号图标把通知标为已读，或者点击组上面的图标把项目中 所有的 通知标为已读。在每个对号图标旁边都有一个静音按钮，你可以点一下，以后就不会收到它相关的通知。

所有这些工具对于处理大量通知非常有用。很多 GitHub 资深用户都关闭邮件通知，在这个页面上处理他们所有的通知。

邮件通知

邮件通知是你处理 GitHub 通知的另一种方式。如果你打开这个选项，每当有通知时，你会收到一封邮件。我们在 Figure 6-13 和 Figure 6-34 看到了一些例子。邮件也会被合适地按话题组织在一起，如果你使用一个具有会话功能的邮件客户端那会很方便。

GitHub 在发送给你的邮件头中附带了很多元数据，这对于设置过滤器和邮件规则非常有帮助。

举个例子，我们来看一看在 Figure 6-34 中发给 Tony 的一封真实邮件的头部，我们会看到下面这些：

```
To: tonychacon/fade <fade@noreply.github.com>
Message-ID: <tonychacon/fade/pull/1@github.com>
Subject: [fade] Wait longer to see the dimming effect better (#1)
X-GitHub-Recipient: tonychacon
List-ID: tonychacon/fade <fade.tonychacon.github.com>
List-Archive: https://github.com/tonychacon/fade
List-Post: <mailto:reply+i-4XXX@reply.github.com>
List-Unsubscribe: <mailto:unsub+i-XXX@reply.github.com>, ...
X-GitHub-Recipient-Address: tchacon@example.com
```

这里有一些有趣的东西。如果你想高亮或者转发这个项目甚至这个合并请求相关的邮件，`Message-ID` 中的信息会以 `<user>/<project>/<type>/<id>` 的格式展现所有的数据。例如，如果这是一个问题 (issue)，那么 `<type>` 字段就会是 “issues” 而不是 “pull” 。

`List-Post` 和 `List-Unsubscribe` 字段表示如果你的邮件客户端能够处理这些，那么你可以很容易地在列表中发贴或取消对这个相关帖子的订阅。那会很有效率，就像在页面中点击静音按钮或在问题 / 合并请求页面点击 “Unsubscribe” 一样。

值得注意的是，如果你同时打开了邮件和网页通知，那么当你在邮件客户端允许加载图片的情况下阅读邮件通知时，对应的网页通知也将会同时被标记为已读。

特殊文件

如果你的版本库中有一些特殊文件，GitHub 会提醒你。

README

第一个就是 `README` 文件，可以是几乎任何 GitHub 可以识别的格式。例如，它可以是 `README` ，`README.md` ，`README.asciidoc` 。如果 GitHub 在你的版本库中找到 README 文件，会把它在项目的首页渲染出来。

很多团队在这个文件里放版本库或项目新人需要了解的所有相关的信息。它一般包含这些内容：

- 该项目的作用
- 如何配置与安装
- 有关如何使用和运行的例子
- 项目的许可证
- 如何向项目贡献力量

因为 GitHub 会渲染这个文件，你可以在文件里植入图片或链接让它更容易理解。

贡献 CONTRIBUTING

另一个 GitHub 可以识别的特殊文件是 `CONTRIBUTING` 。如果你有一个任意扩展名的 `CONTRIBUTING` 文件，当有人开启一个合并请求时 GitHub 会显示 Figure 6-42。

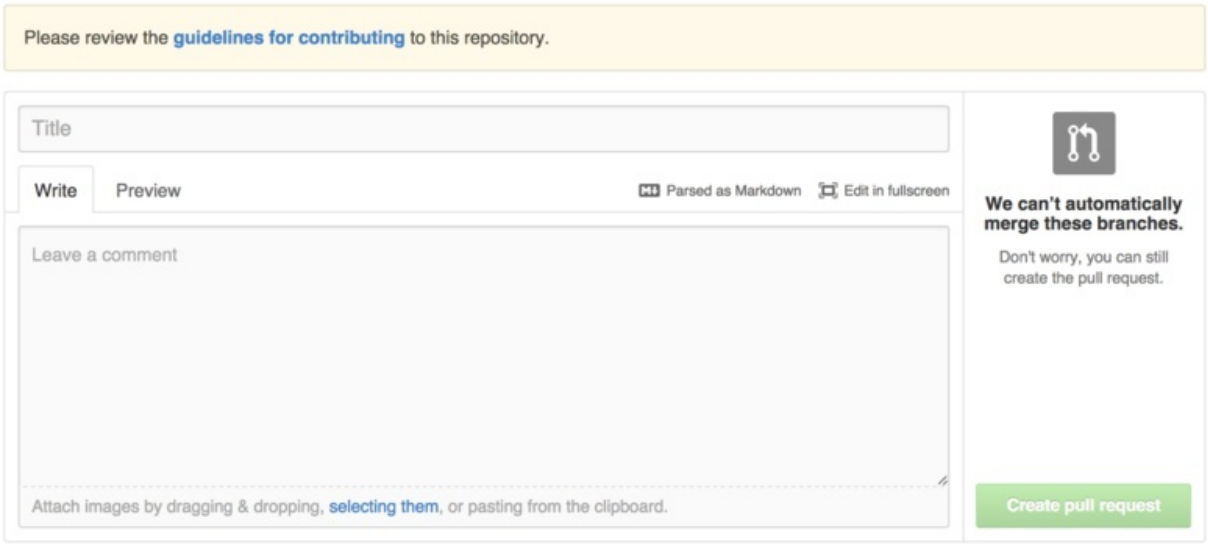


Figure 6-42. 开启合并请求时有 CONTRIBUTING 文件存在.

这个的作用就是你可以在这里指出对于你的项目开启的合并请求你想要的 / 不想要的各种事情。这样别人在开启合并请求之前可以读到这些指导方针。

项目管理

对于一个单个项目其实没有很多管理事务要做，但也有几点有趣的。

改变默认分支

如果你想用 “master” 之外的分支作为你的默认分支，其他人将默认会在这个分支上开启合并请求或进行浏览，你可以在你版本库的设置页面的 “options” 标签下修改。

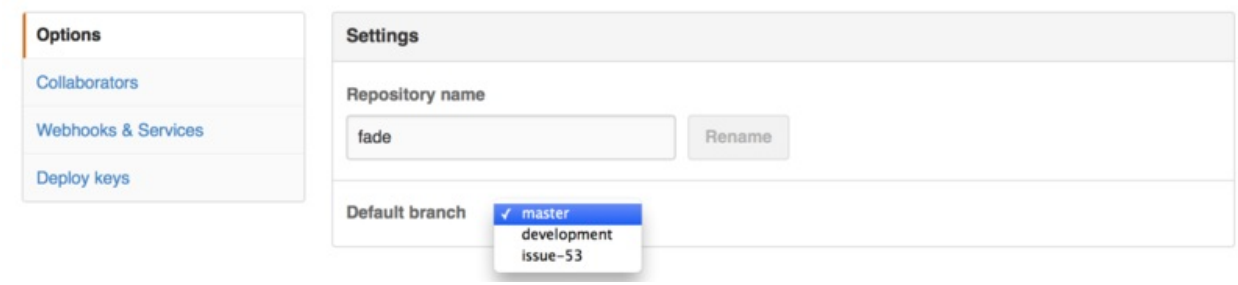


Figure 6-43. 改变项目的默认分支.

简单地改变默认分支下拉列表中的选项，它就会作为所有主要操作的默认分支，他人进行克隆时该分支也将被默认检出。

移交项目

如果你想把一个项目移交给 GitHub 中的另一个人或另一个组织，还是设置页面的这个 " options " 标签下有一个 “Transfer ownership” 选项可以用来干这个。

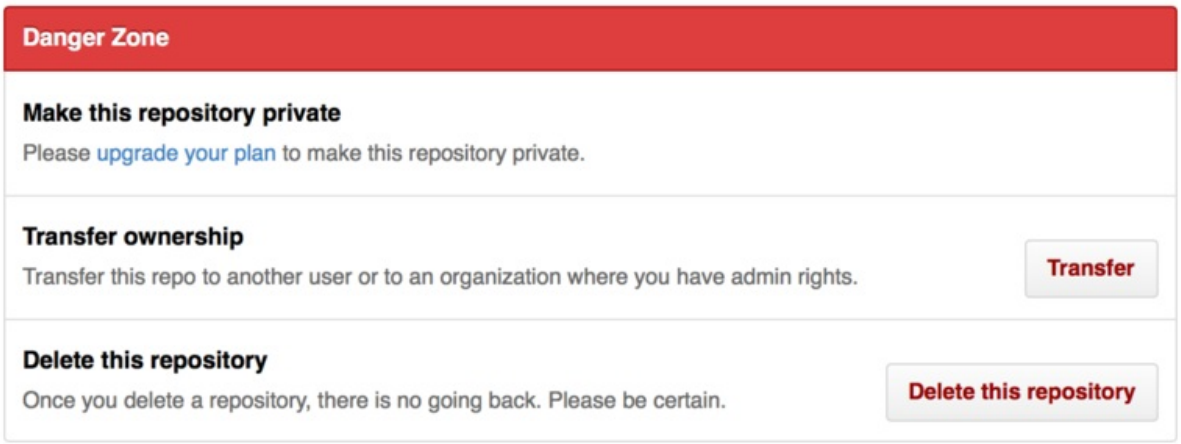


Figure 6-44. 把项目移交给另一个 GitHub 用户或组织。

当你正准备放弃一个项目且正好有别人想要接手时，或者你的项目壮大了想把它移到一个组织里时，这就管用了。

这么做不仅会把版本库连带它所有的观察和星标数都移到另一个地方，它还会将你的 URL 重定向到新的位置。它也重定向了来自 Git 的克隆和抓取，而不仅仅是网页端请求。

6.4 管理组织

管理组织

除了个人帐户之外，GitHub 还提供被称为组织（Organizations）的帐户。 组织账户和个人账户一样都有一个用于存放所拥有项目的命名空间，但是许多其他的东西都是不同的。 组织帐户代表了一组共同拥有多个项目的人，同时也提供一些工具用于对成员进行分组管理。 通常，这种账户被用于开源群组（例如：“perl”或者“rails”），或者公司（例如：“google”或者“twitter”）。

组织的基本知识

我们可以很简单地创建一个组织，只需要点击任意 GitHub 页面右上角的“+”图标，在菜单中选择“New organization”即可。

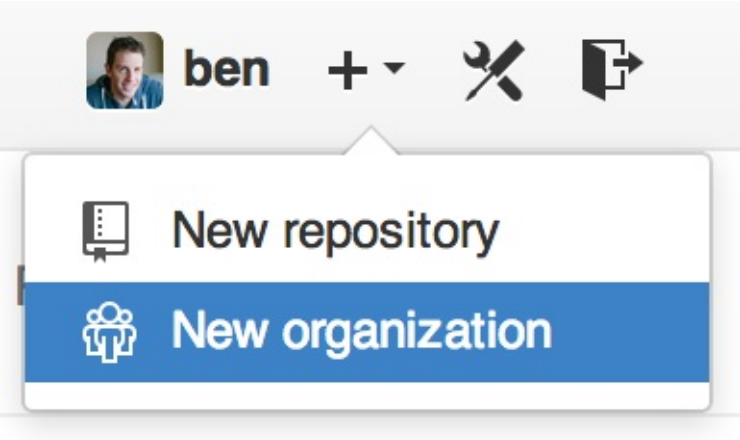


Figure 6-45. “New organization”菜单项

首先你必须提供组织的名称和组织的主要联系邮箱。 然后，如果你希望的话，也可以邀请其他用户作为共同拥有人。

完成以上步骤后，你就会拥有一个全新的组织。 类似于个人帐户，如果组织的所有内容都是开源的，那么你就可以免费使用这个组织。

作为一个组织的拥有者，当你在派生一个版本库的时候，你可以选择把它派生到你的组织的命名空间内。 当你新建版本库时，你可以把它存放到你的个人帐户或你拥有的组织内。 同时，你也会自动地“关注”所有这些组织内的新版本库。

就像头像，你可以为你的组织上传头像，使它更个性化。 同时，也和个人帐户类似，组织会有一个着陆页（landing page），用于列出该组织所有的版本库，并且该页面可供所有人浏览。

下面我们来说一些组织和个人帐户不同的地方。

团队

组织使用团队（Teams）来管理成员，团队就是组织中的一组个人账户和版本库，以及团队成员对这些版本库的访问权限。

例如，假设你的公司有三个版本库：`frontend`、`backend` 和 `deployscripts`。 你会希望你的 HTML/CSS/Javascript 开发者有 `frontend` 或者 `backend` 的访问权限，操作人员有 `backend` 和 `deployscripts` 的访问权限。 团队让这个任务变得更简单，而不用为每个版本库管理它的协作者。

组织页面主要由一个面板（dashboard）构成，这个仪表盘包含了这个组织内的所有版本库，用户和团队。

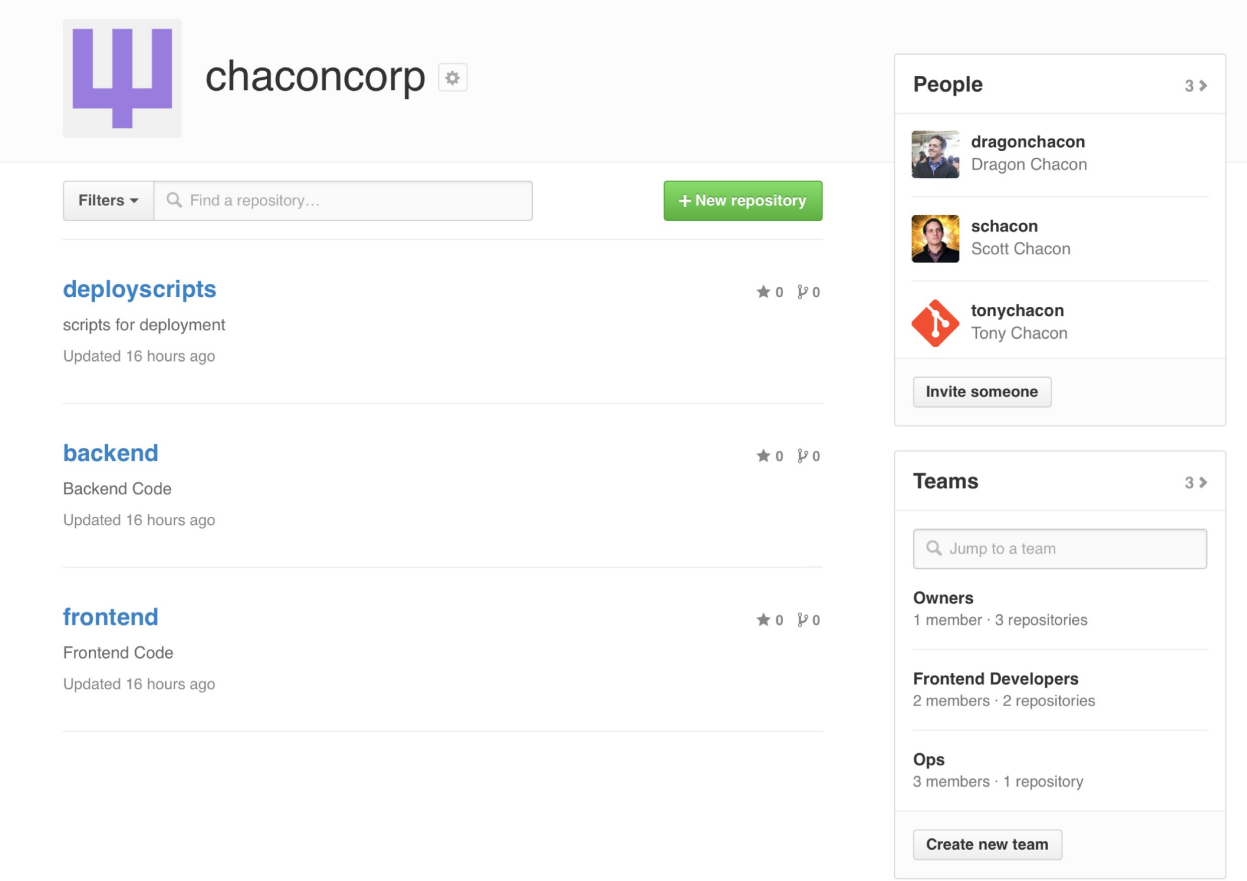


Figure 6-46. 组织页面

你可以点击 Figure 6-46 右边的团队侧边栏（Teams）来管理你的团队。 点击之后，你会进入一个新页面，在这里你可以添加新成员和版本库到团队中，或者管理团队的访问权限和其它设置。 每个团队对于版本库可以有只读、读写和管理三种权限。 你可以通过点击在 Figure 6-47 内的 “Settings” 按钮更改相应权限等级。

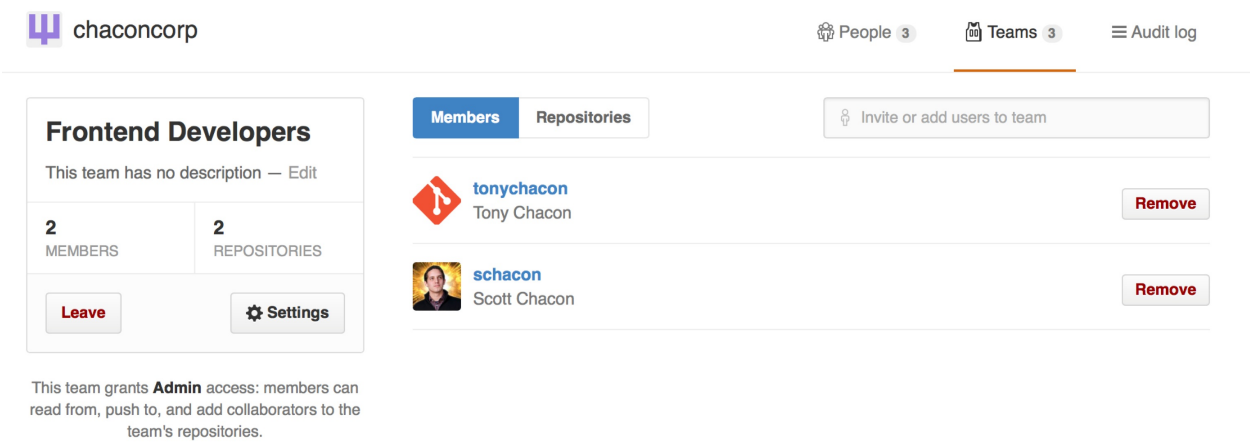


Figure 6-47. 团队页面

当你邀请一个用户加入团队，该用户会收到一封通知他被邀请的邮件。

除此之外，团队也类似于个人帐户，有 `@mentions`（例如：`@acmecorp/frontend`）的功能，不同之处就在于被提及的团队内所有成员都会成为这个话题的订阅者。 当你希望得到团队中某个人的关注，又不知道具体应该问谁的时候，这个功能就显得很有帮助。

一个用户可以加入任意数量的团队，所以别把自己局限于拥有访问控制的团队。 对于某一类课题，像 `ux`，`css` 或者 `refactoring` 这样有着特殊关注点的团队就显得很有帮助，而像 `legal` 和 `colorblind` 这样的就完全是针对它们各自领域的。

审计日志

组织的拥有者还可以访问组织中发生的事情的所有信息。 在 Audit Log 标签页有整个组织的日志，你可以看到谁在世界上哪个地方做了什么事。

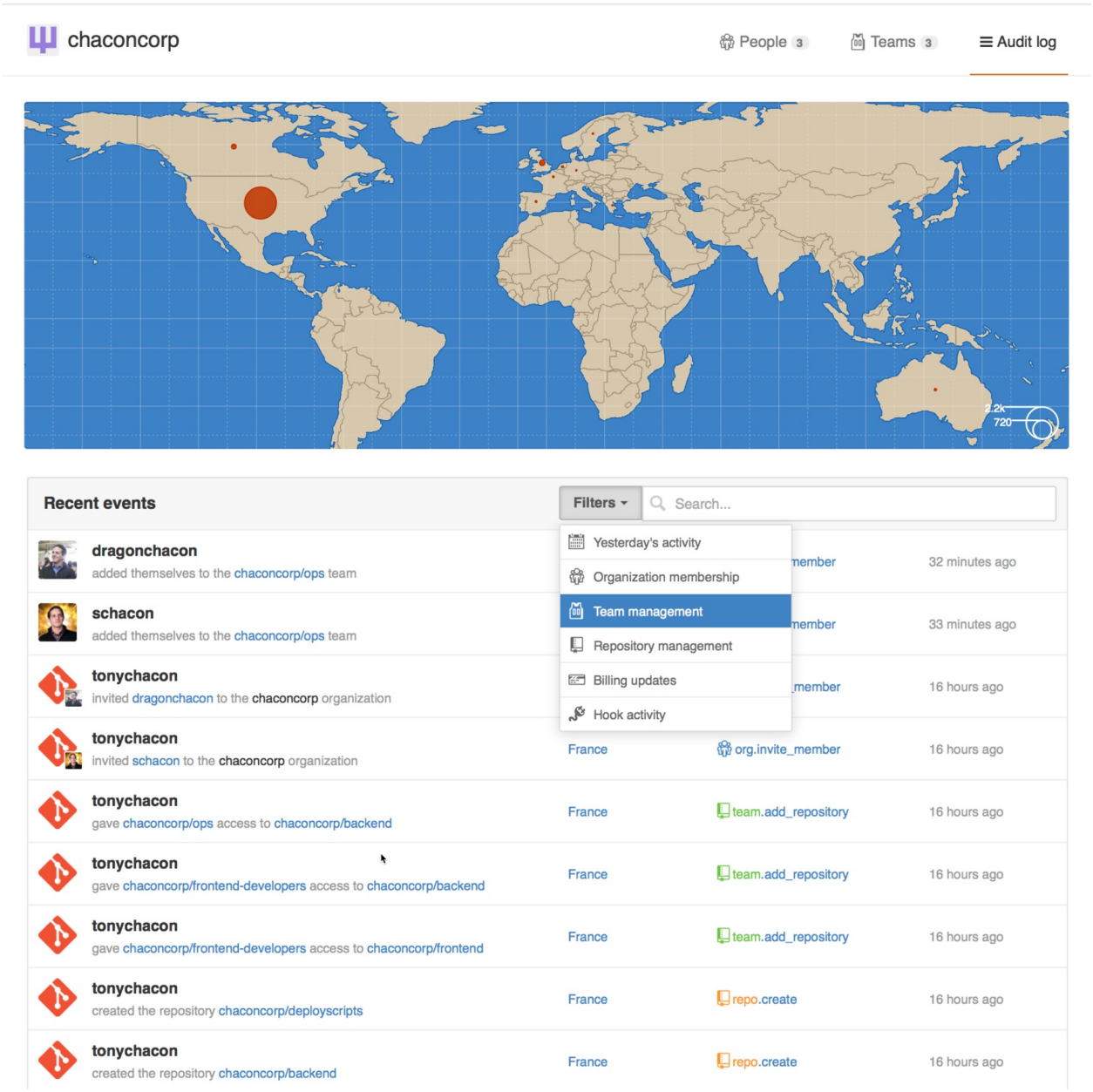


Figure 6-48. 审计日志

你也可以通过选定某一类型的事件、某个地方、某个人对日志进行过滤。

6.5 脚本 GitHub

脚本 GitHub

所以现在我们已经介绍了 GitHub 的大部分功能与工作流程，但是任意一个小组或项目都会去自定义，因为他们想要创造或扩展想要整合的服务。

对我们来说很幸运的是，GitHub 在许多方面都真的很方便 Hack。 在本节中我们将会介绍如何使用 GitHub 钩子系统与 API 接口，使 GitHub 按照我们的设想来工作。

钩子

GitHub 仓库管理中的钩子与服务区块是 GitHub 与外部系统交互最简单的方式。

服务

首先我们来看一下服务。 钩子与服务整合都可以在仓库的设置区块中找到，就在我们之前添加协作者与改变项目的默认分支的地方。 在 “Webhooks and Services” 标签下你会看到与 Figure 6-49类似的内容。

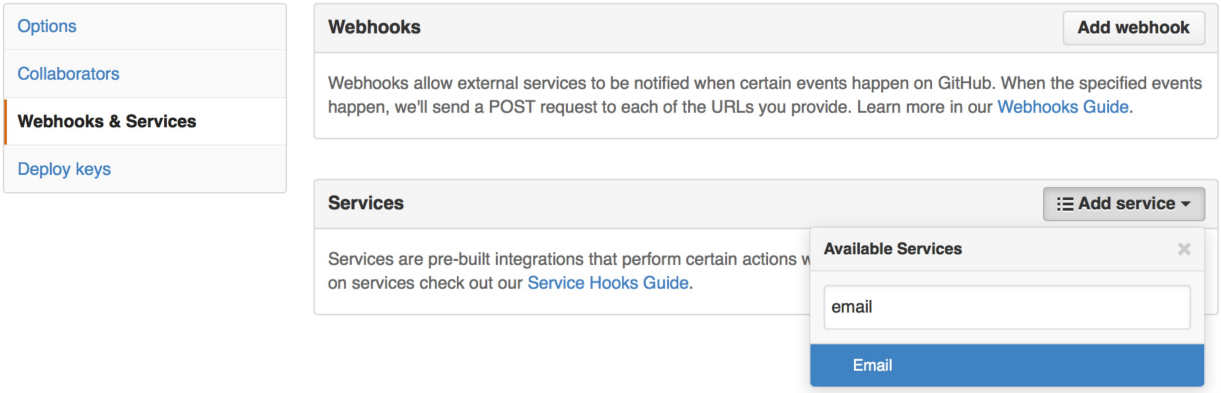


Figure 6-49. 服务与钩子配置区域

有许多可以选择的服务，大多数是整合到其他的商业与开源系统中。 它们中的大多数是为了整合持续集成服务、BUG 与问题追踪系统、聊天室系统与文档系统。 我们将会通过设置一个非常简单的例子来介绍。 如果从 “Add Service” 选择 “email” ，会得到一个类似 Figure 6-50 的配置屏幕。

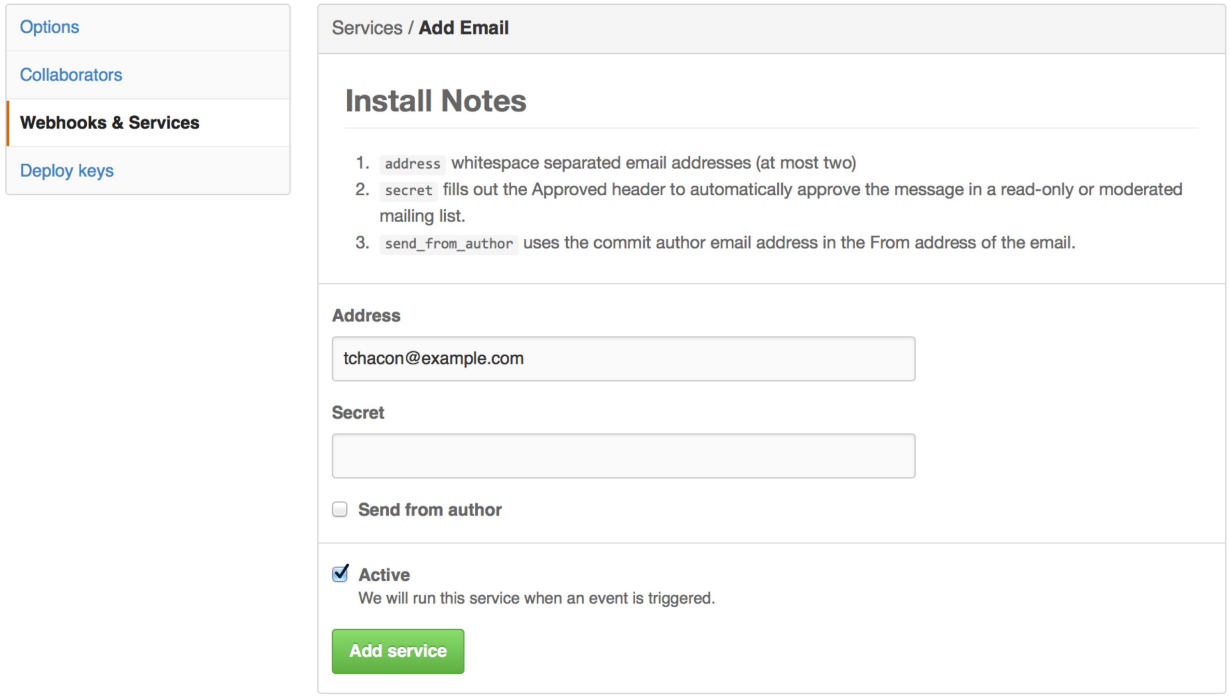


Figure 6-50. 电子邮件服务配置

在本例中，如果我们点击 “Add service” 按钮，每次有人推送内容到仓库时，指定的电子邮件地址都会收到一封邮件。 服务可以监听许多不同类型的事件，但是大多数只监听推送事件然后使用那些数据做一些事情。

如果有一个正在使用的系统想要整合到 GitHub，应当先检查这里看有没有已有的可用的服务整合。 例如，如果正使用 Jenkins 来测试你的代码库，当每次有人推送到你的仓库时你可以启用 Jenkins 内置的整合启动测试运行。

钩子

如果需要做一些更具体的事，或者想要整合一个不在这个列表中的服务或站点，可以转而使用更通用的钩子系统。 GitHub 仓库钩子是非常简单的。 指定一个 URL 然后 GitHub 在任一期望的事件发生时就会发送一个 HTTP 请求到那个 URL 。

通常做这件事的方式是可以设置一个小的 web 服务来监听 GitHub 钩子请求然后使用收到的数据做一些事情。

为了启用一个钩子，点击 Figure 6-49 中的 “Add webhook” 按钮。 这会将你引导至一个类似Figure 6-51 的页面。

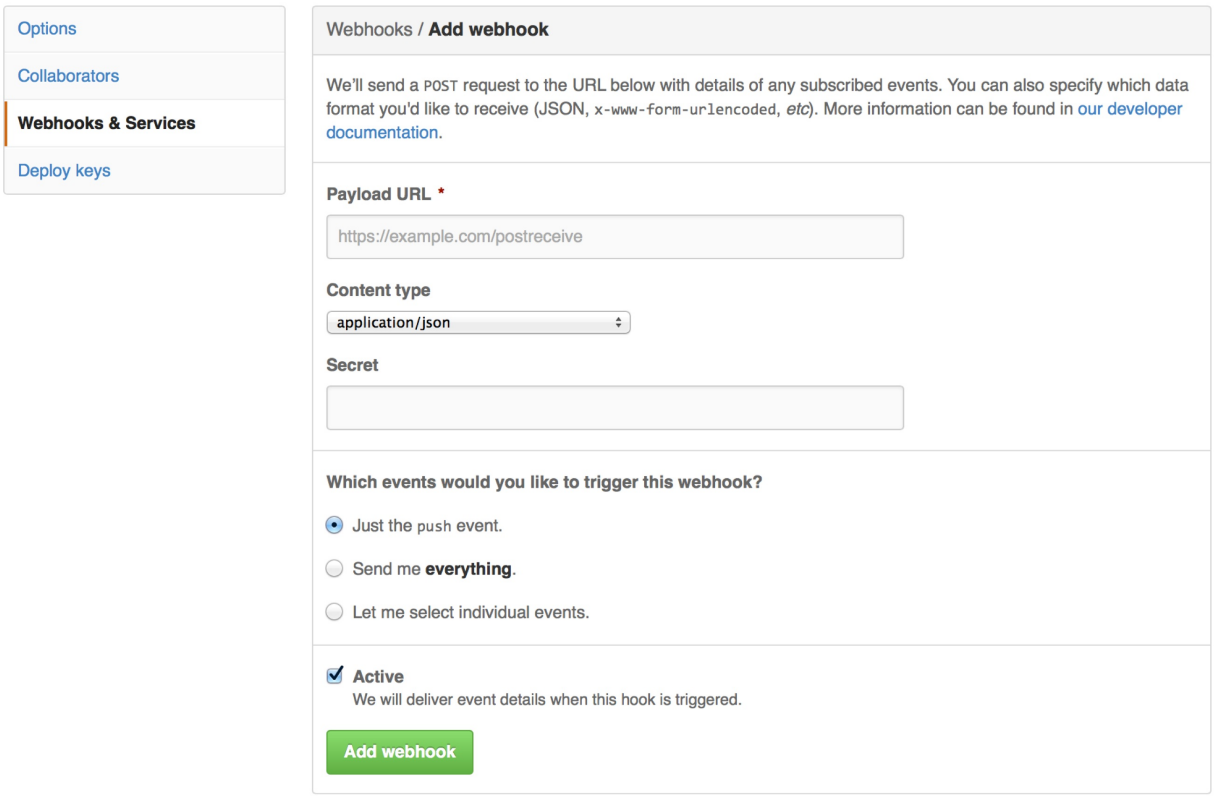


Figure 6-51. Web 钩子配置

Web 钩子的设置非常简单。 大多数情况下只需要输入一个 URL 与一个密钥然后点击 “Add webhook”。 有几个选项可以指定在哪个事件时想要 GitHub 发送请求 — 默认的行为是只有当某人推送新代码到仓库的任一分支时的 `push` 事件获得一个请求。

让我们看一个设置处理 web 钩子的 web 服务的小例子。 我们将会使用 Ruby web 框架 Sinatra，因为它相当简洁，应该能够轻松地看到我们正在做什么。

假设我们想要在某个特定的人推送到我们的项目的特定分支并修改一个特定文件时得到一封邮件。 我们可以相当容易地使用类似下面的代码做到：

```
require 'sinatra'
require 'json'
require 'mail'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON

  # gather the data we're looking for
  pusher = push["pusher"]["name"]
  branch = push["ref"]

  # get a list of all the files touched
  files = push["commits"].map do |commit|
    commit['added'] + commit['modified'] + commit['removed']
```

```
    end
    files = files.flatten.uniq

    # check for our criteria
    if pusher == 'schacon' &&
       branch == 'ref/heads/special-branch' &&
       files.include?('special-file.txt')

      Mail.deliver do
        from     'tchacon@example.com'
        to       'tchacon@example.com'
        subject  'Scott Changed the File'
        body     "ALARM"
      end
    end
  end
```

这里我们拿到一个 GitHub 传送给我们的 JSON 请求然后查找推送者，他们推送到了什么分支以及推送的所有提交都改动了哪些文件。 然后我们检查它是否与我们的条件区配，如果匹配则发送一封邮件。

为了开发与测试类似这样的东西，在设置钩子的地方有一个漂亮的开发者控制台。 可以看到 GitHub 为那个 webhook 的最后几次请求。 对每一个钩子，当它发送后都可以深入挖掘，检测它是否是成功的与请求及回应的消息头与消息体。 这使得测试与调试钩子非常容易。

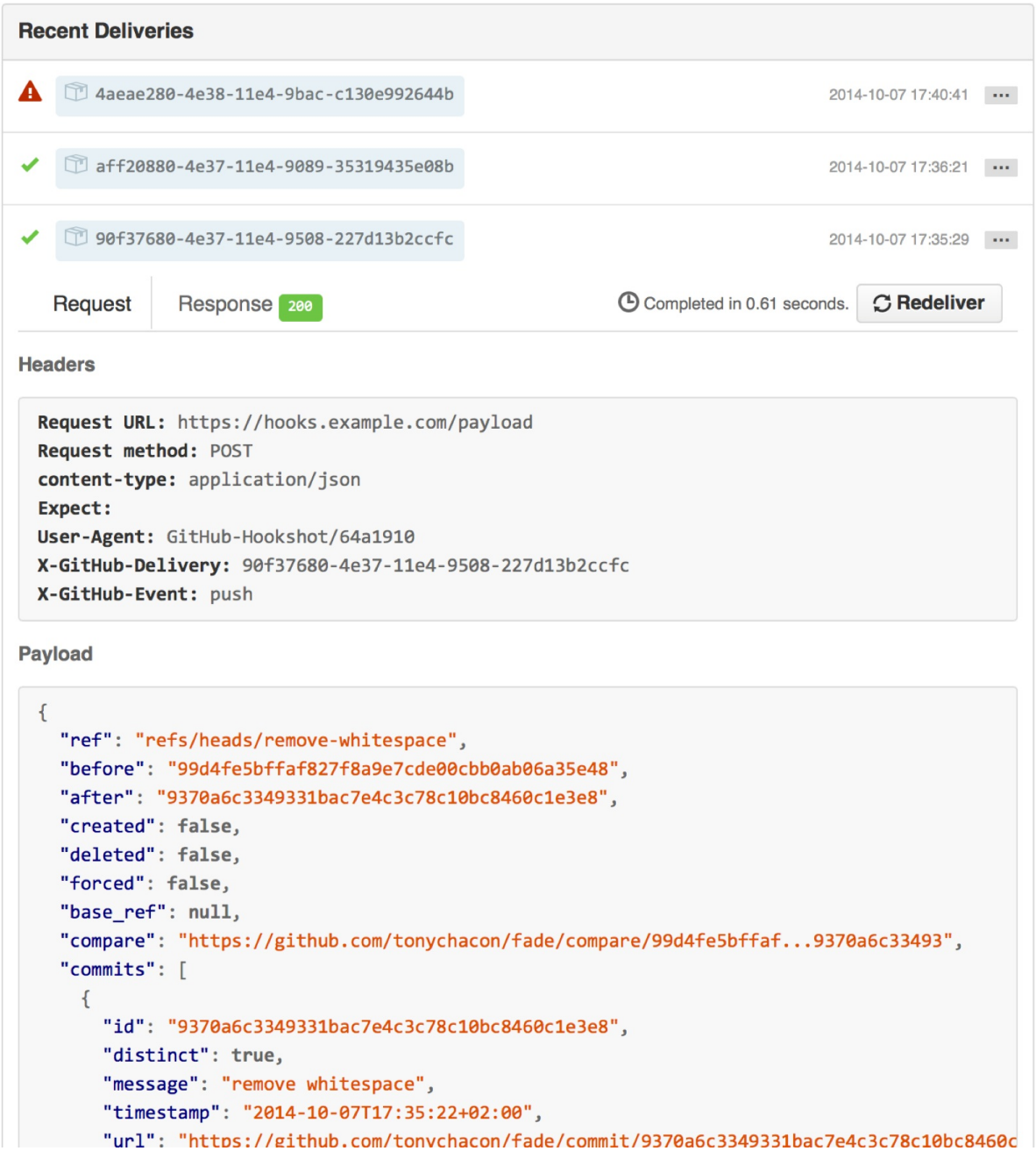


Figure 6-52. Web 钩子调试信息

开发者控制台的另一个很棒的功能是可以轻松地重新发送任何请求来测试你的服务。

关于如何编写 web 钩子与所有可监听的不同事件类型的更多信息，请访问在https://developer.github.com/webhooks/ 的 GitHub 开发者文档。

GitHub API

服务与钩子给你提供了一种方式来接收关于在仓库中发生的事件的推送通知，但是如何获取相关事件的详

情呢？如何自动化一些诸如添加协作者或给问题加标签的事情呢？

这是 GitHub API 派上用场的地方。 在自动化流行的趋势下，GitHub 提供了大量的 API 接口，可以进行几乎任何能在网站上进行的操作。 在本节中我们将会学习如何授权与连接到 API，如何通过 API 在一个问题上评论与如何修改一个 Pull Request 的状态。

基本用途

可以做的最基本的事情是向一个不需要授权的接口上发送一个简单的 GET 请求。 该接口可能是一个用户或开源项目的只读信息。 例如，如果我们想要知道更多关于名为 “schacon” 的用户信息，我们可以运行类似下面的东西：

```
$ curl https://api.github.com/users/schacon
{
  "login": "schacon",
  "id": 70,
  "avatar_url": "https://avatars.githubusercontent.com/u/70",
# …
  "name": "Scott Chacon",
  "company": "GitHub",
  "following": 19,
  "created_at": "2008-01-27T17:19:28Z",
  "updated_at": "2014-06-10T02:37:23Z"
}
```

有大量类似这样的接口来获得关于组织、项目、问题、提交的信息 — 差不多就是你能在 GitHub 上看到的所有东西。 甚至可以使用 API 来渲染任意 Markdown 或寻找一个 `.gitignore` 模板。

```
$ curl https://api.github.com/gitignore/templates/Java
{
  "name": "Java",
  "source": "*.class

# Mobile Tools for Java (J2ME)
.mtj.tmp/

# Package Files #
*.jar
*.war
*.ear

# virtual machine crash logs, see http://www.java.com/en/download/help/error_hotspot.xml
hs_err_pid*
"
}
```

在一个问题上评论

然而，如果想要在网站上进行一个操作，如在 Issue 或 Pull Request 上评论，或者想要查看私有内容或与其交互，你需要授权。

这里提供了几种授权方式。 你可以使用仅需用户名与密码的基本授权，但是通常更好的主意是使用一个个

人访问令牌。 可以从设置页的 “Applications” 标签生成访问令牌。

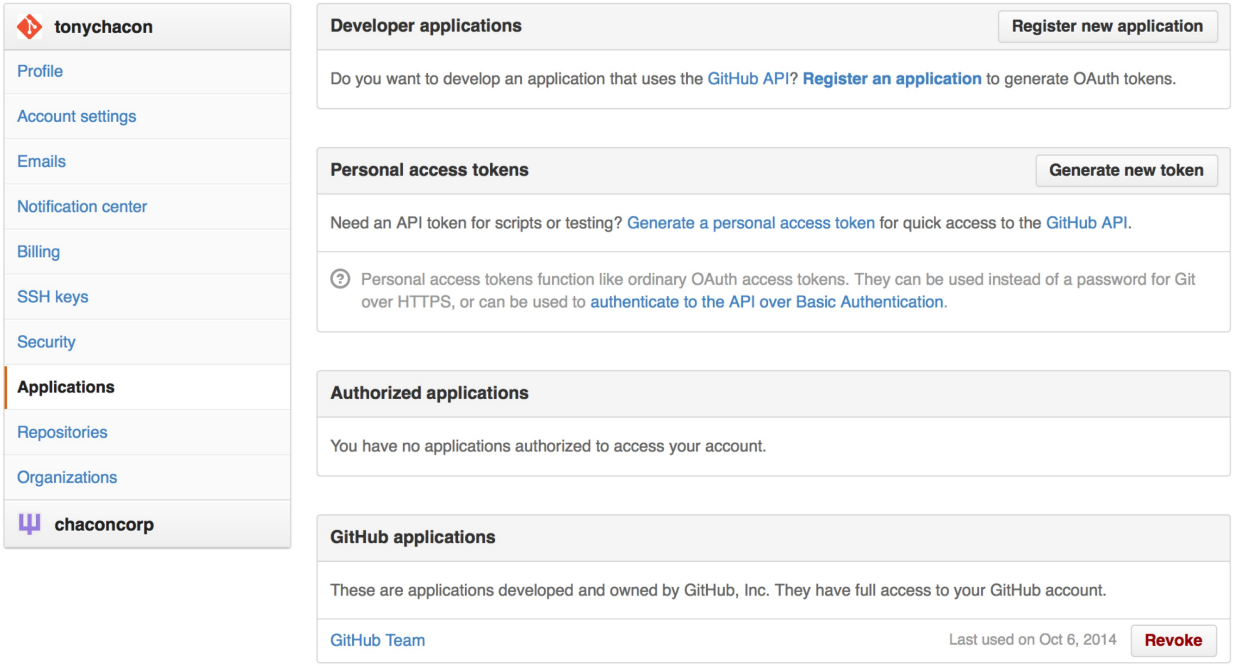


Figure 6-53. 从设置页的 “Applications” 标签生成访问令牌。

它会询问这个令牌的作用域与一个描述。 确保使用一个好的描述信息，这样当脚本或应用不再使用时你会很放心地移除。

GitHub 只会显示令牌一次，所以记得一定要拷贝它。 现在可以在脚本中使用它代替使用用户名写密码来授权。 这很漂亮，因为可以限制想要做的范围并且令牌是可废除的。

这也会有一个提高频率上限的附加优点。 如果没有授权的话，你会被限制在一小时最多发起 60 次请求。 如果授权则可以一小时最多发起 5000 次请求。

所以让我们利用它来对我们的其中一个问题进行评论。 想要对一个特定问题 Issue #6 留下一条评论。 必须使用刚刚生成的令牌作为 Authorization 头信息，发送一个到 `repos/<user>/<repo>/issues/<num>/comments` 的 HTTP POST 请求。

```
$ curl -H "Content-Type: application/json" \
       -H "Authorization: token TOKEN" \
       --data '{"body":"A new comment, :+1:"}' \
       https://api.github.com/repos/schacon/blink/issues/6/comments
{
  "id": 58322100,
  "html_url": "https://github.com/schacon/blink/issues/6#issuecomment-58322100",
  ...
  "user": {
    "login": "tonychacon",
    "id": 7874698,
    "avatar_url": "https://avatars.githubusercontent.com/u/7874698?v=2",
```

```
    "type": "User",
  },
  "created_at": "2014-10-08T07:48:19Z",
  "updated_at": "2014-10-08T07:48:19Z",
  "body": "A new comment, :+1:"
}
```

现在如果进入到那个问题，可以看到我们刚刚发布的评论，像 Figure 6-54 一样。

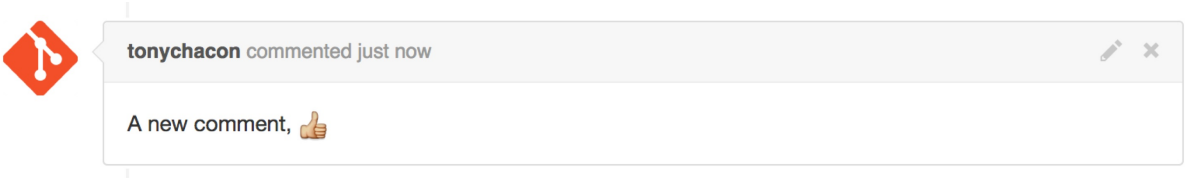


Figure 6-54. 从 GitHub API 发布的一条评论

可以使用 API 去做任何可以在网站上做的事情 — 创建与设置里程碑、指派人员到 Issues 与 Pull Requests，创建与修改标签、访问提交数据、创建新的提交与分支、打开关闭或合并 Pull Requests、创建与编辑团队、在 Pull Request 中评论某行代码、搜索网站等等。

修改 Pull Request 的状态

如果使用 Pull Requests 的话我们将要看到的最后一个例子会很有用。 每一个提交可以有一个或多个与它关联的状态，有 API 来添加与查询状态。

大多数持续集成与测试服务通过测试推送的代码后使用这个 API 来回应，然后报告提交是否通过了全部测试。 你也可以使用该接口来检查提交信息是否经过合适的格式化、提交者是否遵循了所有你的贡献准则、提交是否经过有效的签名 — 种种这类事情。

假设在仓库中设置了一个 web 钩子访问一个用来检查提交信息中的 `Signed-off-by` 字符串的小的 web 服务。

```
require 'httparty'
require 'sinatra'
require 'json'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON
  repo_name = push['repository']['full_name']

  # look through each commit message
  push["commits"].each do |commit|

    # look for a Signed-off-by string
    if /Signed-off-by/.match commit['message']
      state = 'success'
      description = 'Successfully signed off!'
    else
      state = 'failure'
      description = 'No signoff found.'
    end
```

```
    # post status to GitHub
    sha = commit["id"]
    status_url = "https://api.github.com/repos/#{repo_name}/statuses/#{sha}"

    status = {
      "state"       => state,
      "description" => description,
      "target_url"  => "http://example.com/how-to-signoff",
      "context"     => "validate/signoff"
    }
    HTTParty.post(status_url,
      :body => status.to_json,
      :headers => {
        'Content-Type'  => 'application/json',
        'User-Agent'    => 'tonychacon/signoff',
        'Authorization' => "token #{ENV['TOKEN']}" }
    )
  end
end
```

希望这相当容易做。 在这个 web 钩子处理器中我们浏览刚刚推送上来的每一个提交，在提交信息中查找字符串 Signed-off-by 并且最终使用 HTTP 向 `/repos/<user>/<repo>/statuses/<commit_sha>` API 接口发送一个带有状态的 POST 请求。

在本例中可以发送一个状态（success, failure, error）、一个发生了什么的描述信息、一个用户可以了解更多信息的目标 URL 与一个 “context” 以防一个单独的提交有多个状态。 例如，一个测试服务可以提供一个状态与一个类似这样的验证服务也可能提供一个状态 — “context” 字段是用来区别它们的。

如果某人在 GitHub 中打开了一个新的 Pull Request 并且这个钩子已经设置，会看到类似Figure 6-55 的信息。

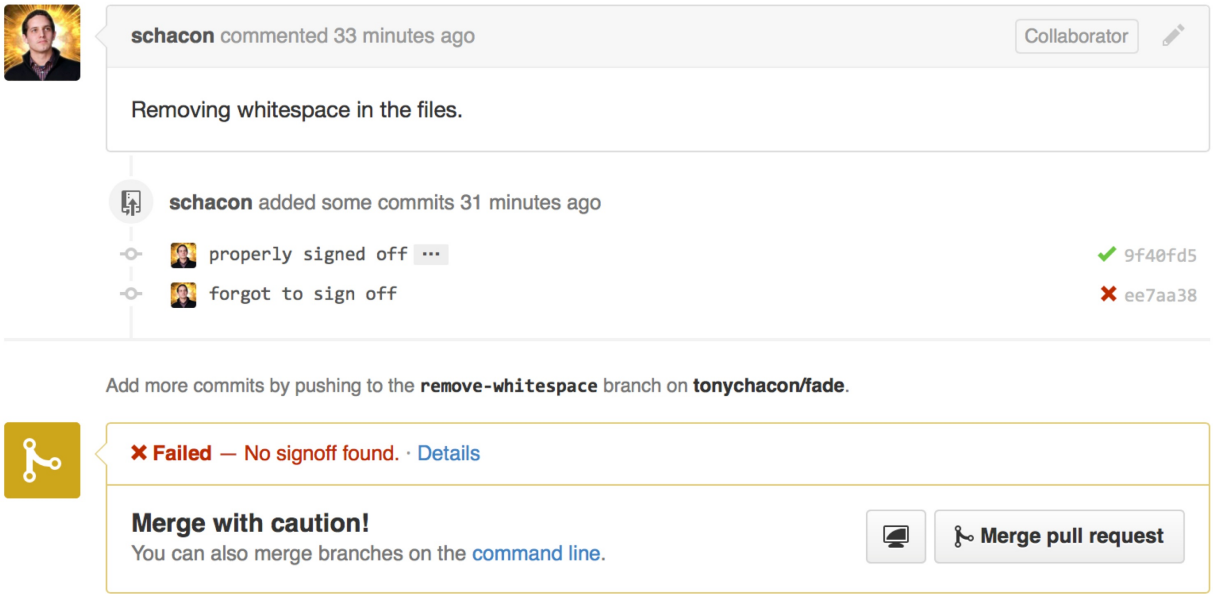


Figure 6-55. 通过 API 的提交状态

现在可以看到一个小的绿色对勾标记在提交信息中有 “Signed-off-by” 的提交旁边，红色的对勾标记在作者忘记签名的提交旁边。 也可以看到 Pull Request 显示在那个分支上的最后提交的状态，如果失败的话会警告你。 如果对测试结果使用这个 API 那么就不会不小心合并某些未通过测试的最新提交。

Octokit

尽管我们在这些例子中都是通过 `curl` 与基本的 HTTP 请求来做几乎所有的事情，还有一些以更自然的方式利用 API 的开源库存在着。 在写这篇文章的时候，被支持的语言包括 Go、Objective-C、Ruby 与 .NET。 访问 http://github.com/octokit 了解更多相关信息，它们帮你处理了更多 HTTP 相关的内容。

希望这些工具能帮助你自定义与修改 GitHub 来更好地为特定的工作流程工作。 关于全部 API 的完整文档与常见任务的指南，请查阅 https://developer.github.com。

6.6 总结

总结

现在你已经是一名 GitHub 用户了。 你知道了如何创建账户、管理组织、创建和推送版本库、向别人的项目提供贡献以及接受别人的贡献。 在下一章中，你将学习更多强有力的工具，以及处理复杂情况的知识，这些将使你成为真正的 Git 大师。

7. Git 工具

第7章. Git 工具

现在，你已经学习了管理或者维护 Git 仓库、实现代码控制所需的大多数日常命令和工作流程。你已经尝试了跟踪和提交文件的基本操作，并且发挥了暂存区和轻量级的分支及合并的威力。

接下来你将学习一些 Git 的强大功能，这些功能你可能并不会在日常操作中使用，但在某些时候你可能会需要。

7.1 选择修订版本

选择修订版本

Git 允许你通过几种方法来指明特定的或者一定范围内的提交。 了解它们并不是必需的，但是了解一下总没坏处。

单个修订版本

你可以通过 Git 给出的 SHA-1 值来获取一次提交，不过还有很多更人性化的方式来做同样的事情。 本节将会介绍获取单个提交的多种方法。

简短的 SHA-1

Git 十分智能，你只需要提供 SHA-1 的前几个字符就可以获得对应的那次提交，当然你提供的 SHA-1 字符数量不得少于 4 个，并且没有歧义——也就是说，当前仓库中只有一个对象以这段 SHA-1 开头。

例如查看一次指定的提交，假设你执行 `git log` 命令来查看之前新增一个功能的那次提交:

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff
```

假设这个提交是 `1c002dd....`，如果你想 `git show` 这个提交，下面的命令是等价的（假设简短的版本没有歧义）：

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Git 可以为 SHA-1 值生成出简短且唯一的缩写。 如果你在 `git log` 后加上 `--abbrev-commit` 参数，输出结果里就会显示简短且唯一的值；默认使用七个字符，不过有时为了避免 SHA-1 的歧义，会增

加字符数：

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit
```

通常 8 到 10 个字符就已经足够在一个项目中避免 SHA-1 的歧义。

比如 Linux 内核这个相当大的 Git 项目，目前有超过 45 万个提交，包含 360 万个对象，也只需要前 11 个字符就能保证唯一性。

> NOTE

> ## 关于 SHA-1 的简短说明

> 许多人觉得他们的仓库里有可能出现两个 SHA-1 值相同的对象。 然后呢？

> 如果你真的向仓库里提交了一个跟之前的某个对象具有相同 SHA-1 值的对象，Git 发现仓库里已经存在了拥有相同 HASH 值的对象，就会认为这个新的提交是已经被写入仓库的。 如果之后你想检出那个对象时，你将得到先前那个对象的数据。

> 但是这种情况发生的概率十分渺小。 SHA-1 摘要长度是 20 字节，也就是 160 位。 280 个随机哈希对象才有 50% 的概率出现一次冲突（计算冲突机率的公式是 `p = (n(n-1)/2) * (1/2^160))`）。280 是 1.2 x 10^24 也就是一亿亿亿。 那是地球上沙粒总数的 1200 倍。

> 举例说一下怎样才能产生一次 SHA-1 冲突。 如果地球上 65 亿个人类都在编程，每人每秒都在产生等价于整个 Linux 内核历史（360 万个 Git 对象）的代码，并将之提交到一个巨大的 Git 仓库里面，这样持续两年的时间才会产生足够的对象，使其拥有 50% 的概率产生一次 SHA-1 对象冲突。 这要比你编程团队的成员同一个晚上在互不相干的意外中被狼袭击并杀死的机率还要小。

分支引用

指明一次提交最直接的方法是有一个指向它的分支引用。 这样你就可以在任意一个 Git 命令中使用这个分支名来代替对应的提交对象或者 SHA-1 值。 例如，你想要查看一个分支的最后一次提交的对象，假设 `topic1` 分支指向 `ca82a6d` ，那么以下的命令是等价的:

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

如果你想知道某个分支指向哪个特定的 SHA-1，或者想看任何一个例子中被简写的 SHA-1 ，你可以使用一个叫做 `rev-parse` 的 Git 探测工具。 你可以在 Git 内部原理 中查看更多关于探测工具的信息。简单来说， `rev-parse` 是为了底层操作而不是日常操作设计的。 不过，有时你想看 Git 现在到底处于什么状态时，它可能会很有用。 你可以在你的分支上执行 `rev-parse`

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

引用日志

当你在工作时， Git 会在后台保存一个引用日志(reflog)，引用日志记录了最近几个月你的 HEAD 和分支引用所指向的历史。

你可以使用 `git reflog` 来查看引用日志

```
$ git reflog
734713b HEAD@{0}: commit: fixed refs handling, added gc auto, updated
d921970 HEAD@{1}: merge phedders/rdocs: Merge made by recursive.
1c002dd HEAD@{2}: commit: added some blame and merge stuff
1c36188 HEAD@{3}: rebase -i (squash): updating HEAD
95df984 HEAD@{4}: commit: # This is a combination of two commits.
1c36188 HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5 HEAD@{6}: rebase -i (pick): updating HEAD
```

每当你的 HEAD 所指向的位置发生了变化，Git 就会将这个信息存储到引用日志这个历史记录里。 通过这些数据，你可以很方便地获取之前的提交历史。 如果你想查看仓库中 HEAD 在五次前的所指向的提交，你可以使用 `@{n}` 来引用 reflog 中输出的提交记录。

```
$ git show HEAD@{5}
```

你同样可以使用这个语法来查看某个分支在一定时间前的位置。 例如，查看你的 `master` 分支在昨天的时候指向了哪个提交，你可以输入

```
$ git show master@{yesterday}
```

就会显示昨天该分支的顶端指向了哪个提交。 这个方法只对还在你引用日志里的数据有用，所以不能用来查好几个月之前的提交。

可以运行 `git log -g` 来查看类似于 `git log` 输出格式的引用日志信息：

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: fixed refs handling, added gc auto, updated
```

```
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

值得注意的是，引用日志只存在于本地仓库，一个记录你在你自己的仓库里做过什么的日志。 其他人拷贝的仓库里的引用日志不会和你的相同；而你新克隆一个仓库的时候，引用日志是空的，因为你在仓库里还没有操作。 `git show HEAD@{2.months.ago}` 这条命令只有在你克隆了一个项目至少两个月时才会有用——如果你是五分钟前克隆的仓库，那么它将不会有结果返回。

祖先引用

祖先引用是另一种指明一个提交的方式。 如果你在引用的尾部加上一个 `^` ， Git 会将其解析为该引用的上一个提交。 假设你的提交历史是：

```
$ git log --pretty=format:'%h %s' --graph
* 734713b fixed refs handling, added gc auto, updated tests
*   d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd added some blame and merge stuff
|/
* 1c36188 ignore *.gem
* 9b29157 add open3_detach to gemspec file list
```

你可以使用 `HEAD^` 来查看上一个提交，也就是 “HEAD 的父提交” ：

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

你也可以在 `^` 后面添加一个数字——例如 `d921970^2` 代表 “d921970 的第二父提交” 这个语法只适用于合并(merge)的提交，因为合并提交会有多个父提交。 第一父提交是你合并时所在分支，而第二父提交是你所合并的分支：

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800
```

```
    added some blame and merge stuff

$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date:   Wed Dec 10 22:22:03 2008 +0000

    Some rdoc changes
```

另一种指明祖先提交的方法是 `~` 。 同样是指向第一父提交，因此 `HEAD~` 和 `HEAD^` 是等价的。 而区别在于你在后面加数字的时候。 `HEAD~2` 代表 “第一父提交的第一父提交” ，也就是 “祖父提交” —— Git 会根据你指定的次数获取对应的第一父提交。 例如，在之前的列出的提交历史中， `HEAD~3` 就是

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

也可以写成 `HEAD^^^` ，也是第一父提交的第一父提交的第一父提交：

```
$ git show HEAD^^^
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

你也可以组合使用这两个语法 —— 你可以通过 `HEAD~3^2` 来取得之前引用的第二父提交（假设它是一个合并提交）。

提交区间

你已经学会如何单次的提交，现在来看看如何指明一定区间的提交。 当你有很多分支时，这对管理你的分支时十分有用，你可以用提交区间来解决 “这个分支还有哪些提交尚未合并到主分支？” 的问题

双点

最常用的指明提交区间语法是双点。 这种语法可以让 Git 选出在一个分支中而不在另一个分支中的提交。 例如，你有如下的提交历史 Figure 7-1

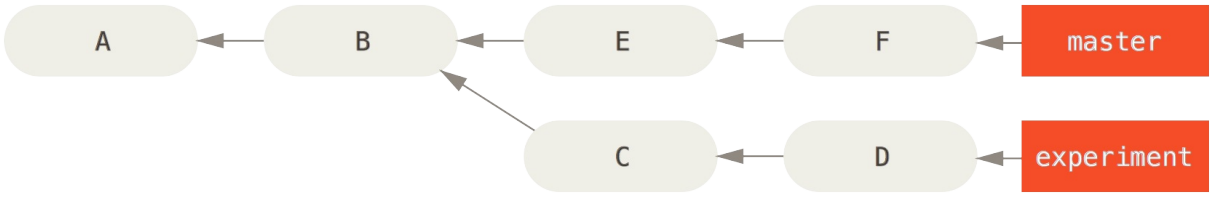


Figure 7-1. Example history for range selection.

你想要查看 experiment 分支中还有哪些提交尚未被合并入 master 分支。 你可以使用 `master..experiment` 来让 Git 显示这些提交。也就是 “在 experiment 分支中而不在 master 分支中的提交”。 为了使例子简单明了，我使用了示意图中提交对象的字母来代替真实日志的输出，所以会显示：

```
$ git log master..experiment
D
C
```

反过来，如果你想查看在 `master` 分支中而不在 `experiment` 分支中的提交，你只要交换分支名即可。 `experiment..master` 会显示在 `master` 分支中而不在 `experiment` 分支中的提交：

```
$ git log experiment..master
F
E
```

这可以让你保持 `experiment` 分支跟随最新的进度以及查看你即将合并的内容。 另一个常用的场景是查看你即将推送到远端的内容：

```
$ git log origin/master..HEAD
```

这个命令会输出在你当前分支中而不在远程 `origin` 中的提交。 如果你执行了 `git push` 并且你的当前分支正在跟踪 `origin/master`，`git log origin/master..HEAD` 所输出的提交将会被传输到远端服务器。 如果你留空了其中的一边， Git 会默认为 HEAD。 例如，`git log origin/master..` 将会输出与之前例子相同的结果 —— Git 使用 HEAD 来代替留空的一边。

多点

双点语法很好用，但有时候你可能需要两个以上的分支才能确定你所需要的修订，比如查看哪些提交是被包含在某些分支中的一个，但是不在你当前的分支上。 Git 允许你在任意引用前加上 `^` 字符或者 `--not` 来指明你不希望提交被包含其中的分支。 因此下列3个命令是等价的：

```
$ git log refA..refB
```

```
$ git log ^refA refB
$ git log refB --not refA
```

这个语法很好用，因为你可以在查询中指定超过两个的引用，这是双点语法无法实现的。 比如，你想查看所有被 `refA` 或 `refB` 包含的但是不被 `refC` 包含的提交，你可以输入下面中的任意一个命令

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

这就构成了一个十分强大的修订查询系统，你可以通过它来查看你的分支里包含了哪些东西。

三点

最后一种主要的区间选择语法是三点，这个语法可以选择出被两个引用中的一个包含但又不被两者同时包含的提交。 再看看之前双点例子中的提交历史。 如果你想看 `master` 或者 `experiment` 中包含的但不是两者共有的提交，你可以执行

```
$ git log master...experiment
F
E
D
C
```

这和通常 `log` 按日期排序的输出一样，仅仅给出了4个提交的信息。

这种情形下，`log` 命令的一个常用参数是 `--left-right` ，它会显示每个提交到底处于哪一侧的分支。 这会让输出数据更加清晰。

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

有了这些工具，你就可以十分方便地查看你 Git 仓库中的提交。

7.2 交互式暂存

交互式暂存

Git 自带的一些脚本可以使在命令行下工作更容易。 本节的几个互交命令可以帮助你将文件的特定部分组合成提交。 当你修改一组文件后，希望这些改动能放到若干提交而不是混杂在一起成为一个提交时，这几个工具会非常有用。 通过这种方式，可以确保提交是逻辑上独立的变更集，同时也会使其他开发者在与你工作时很容易地审核。 如果运行 `git add` 时使用 `-i` 或者 `--interactive` 选项，Git 将会进入一个交互式终端模式，显示类似下面的东西：

```
$ git add -i
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb

*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now>
```

可以看到这个命令以非常不同的视图显示了暂存区 - 基本上与 `git status` 是相同的信息，但是更简明扼要一些。 它将暂存的修改列在左侧，未暂存的修改列在右侧。

在这块区域后是命令区域。 在这里你可以做一些工作，包括暂存文件、取消暂存文件、暂存文件的一部分、添加未被追踪的文件、查看暂存内容的区别。

暂存与取消暂存文件

如果在 `What now>` 提示符后键入 `2` 或 `u`，脚本将会提示想要暂存哪个文件：

```
What now> 2
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Update>>
```

要暂存 TODO 与 index.html 文件，可以输入数字：

```
Update>> 1,2
           staged     unstaged path
* 1:    unchanged        +0/-1 TODO
* 2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Update>>
```

每个文件前面的 `*` 意味着选中的文件将会被暂存。 如果在 `Update>>` 提示符后不输入任何东西并直接按回车，Git 将会暂存之前选择的文件：

```
Update>>
updated 2 paths

*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 1
           staged     unstaged path
  1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

现在可以看到 TODO 与 index.html 文件已经被暂存而 simplegit.rb 文件还未被暂存。 如果这时想要取消暂存 TODO 文件，使用 `3` 或 `r`（撤消）选项：

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 3
           staged     unstaged path
  1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Revert>> 1
           staged     unstaged path
* 1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Revert>> [enter]
reverted one path
```

再次查看 Git 状态，可以看到已经取消暂存 TODO 文件：

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 1
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

如果想要查看已暂存内容的区别，可以使用 `6` 或 `d`（区别）命令。 它会显示暂存文件的一个列表，可以从中选择想要查看的暂存区别。 这跟你在命令行指定 `git diff --cached` 非常相似：

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
```

```
What now> 6
           staged     unstaged path
  1:        +1/-1      nothing index.html
Review diff>> 1
diff --git a/index.html b/index.html
index 4d07108..4335f49 100644
--- a/index.html
+++ b/index.html
@@ -16,7 +16,7 @@ Date Finder

 <p id="out">...</p>

-<div id="footer">contact : support@github.com</div>
+<div id="footer">contact : email.support@github.com</div>

 <script type="text/javascript">
```

通过这些基本命令，可以使用交互式添加模式来轻松地处理暂存区。

暂存补丁

Git 也可以暂存文件的特定部分。 例如，如果在 simplegit.rb 文件中做了两处修改，但只想要暂存其中的一个而不是另一个，Git 会帮你轻松地完成。 从交互式提示符中，输入 `5` 或 `p`（补丁）。 Git 会询问你想要部分暂存哪些文件；然后，对已选择文件的每一个部分，它都会一个个地显示文件区别并询问你是否想要暂存它们：

```
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index dd5ecc4..57399e0 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -22,7 +22,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log -n 25 #{treeish}")
+    command("git log -n 30 #{treeish}")
   end

   def blame(path)
Stage this hunk [y,n,a,d,/,j,J,g,e,?]?
```

这时有很多选项。 输入 `?` 显示所有可以使用的命令列表：

```
Stage this hunk [y,n,a,d,/,j,J,g,e,?]? ?
y - stage this hunk
n - do not stage this hunk
a - stage this and all the remaining hunks in the file
d - do not stage this hunk nor any of the remaining hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
```

```
  ? - print help
```

通常情况下可以输入 y 或 n 来选择是否要暂存每一个区块，当然，暂存特定文件中的所有部分或为之后的选择跳过一个区块也是非常有用的。 如果你只暂存文件的一部分，状态输出可能会像下面这样：

```
What now> 1
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:        +1/-1      nothing index.html
  3:        +1/-1        +4/-0 lib/simplegit.rb
```

simplegit.rb 文件的状态很有趣。 它显示出若干行被暂存与若干行未被暂存。 已经部分地暂存了这个文件。 在这时，可以退出交互式添加脚本并且运行 `git commit` 来提交部分暂存的文件。

也可以不必在交互式添加模式中做部分文件暂存 - 可以在命令行中使用 `git add -p` 或 `git add --patch` 来启动同样的脚本。

更进一步地，可以使用 `reset --patch` 命令的补丁模式来部分重置文件，通过 `checkout --patch` 命令来部分检出文件与 `stash save --patch` 命令来部分暂存文件。 我们将会在接触这些命令的高级使用方法时了解更多详细信息。

7.3 储藏与清理

储藏与清理

有时，当你在项目的一部分上已经工作一段时间后，所有东西都进入了混乱的状态，而这时你想要切换到另一个分支做一点别的事情。 问题是，你不想仅仅因为过会儿回到这一点而为做了一半的工作创建一次提交。 针对这个问题的答案是 `git stash` 命令。

储藏会处理工作目录的脏的状态 - 即，修改的跟踪文件与暂存改动 - 然后将未完成的修改保存到一个栈上，而你可以在任何时候重新应用这些改动。

储藏工作

为了演示，进入项目并改动几个文件，然后可能暂存其中的一个改动。 如果运行 `git status` ，可以看到有改动的状态：

```
$ git status
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   lib/simplegit.rb
```

现在想要切换分支，但是还不想要提交之前的工作；所以储藏修改。 将新的储藏推送到栈上，运行 `git stash` 或 `git stash save` ：

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

工作目录是干净的了：

```
$ git status
# On branch master
nothing to commit, working directory clean
```

在这时，你能够轻易地切换分支并在其他地方工作；你的修改被存储在栈上。 要查看储藏的东西，可以使用 `git stash list` ：

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

在本例中，有两个之前做的储藏，所以你接触到了三个不同的储藏工作。 可以通过原来 stash 命令的帮助提示中的命令将你刚刚储藏的工作重新应用：`git stash apply`。 如果想要应用其中一个更旧的储藏，可以通过名字指定它，像这样：`git stash apply stash@{2}`。 如果不指定一个储藏，Git 认为指定的是最近的储藏：

```
$ git stash apply
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   index.html
#      modified:   lib/simplegit.rb
#
```

可以看到 Git 重新修改了当你保存储藏时撤消的文件。 在本例中，当尝试应用储藏时有一个干净的工作目录，并且尝试将它应用在保存它时所在的分支；但是有一个干净的工作目录与应用在同一分支并不是成功应用储藏的充分必要条件。 可以在一个分支上保存一个储藏，切换到另一个分支，然后尝试重新应用这些修改。 当应用储藏时工作目录中也可以有修改与未提交的文件 - 如果有任何东西不能干净地应用，Git 会产生合并冲突。

文件的改动被重新应用了，但是之前暂存的文件却没有重新暂存。 想要那样的话，必须使用 `--index` 选项来运行 `git stash apply` 命令，来尝试重新应用暂存的修改。 如果已经那样做了，那么你将回到原来的位置：

```
$ git stash apply --index
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      modified:   index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   lib/simplegit.rb
#
```

应用选项只会尝试应用暂存的工作 - 在堆栈上还有它。 可以运行 `git stash drop` 加上将要移除的储藏的名字来移除它：

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
```

```
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

也可以运行 `git stash pop` 来应用储藏然后立即从栈上扔掉它。

创造性的储藏

有几个储藏的变种可能也很有用。 第一个非常流行的选项是 `stash save` 命令的 `--keep-index` 选项。 它告诉 Git 不要储藏任何你通过 git add 命令已暂存的东西。

当你做了几个改动并只想提交其中的一部分，过一会儿再回来处理剩余改动时，这个功能会很有用。

```
$ git status -s
M  index.html
 M lib/simplegit.rb

$ git stash --keep-index
Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file

$ git status -s
M  index.html
```

另一个经常使用储藏来做的事情是像储藏跟踪文件一样储藏未跟踪文件。 默认情况下，`git stash` 只会储藏已经在索引中的文件。 如果指定 `--include-untracked` 或 `-u` 标记，Git 也会储藏任何创建的未跟踪文件。

```
$ git status -s
M  index.html
 M lib/simplegit.rb
?? new-file.txt

$ git stash -u
Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file

$ git status -s
$
```

最终，如果指定了 `--patch` 标记，Git 不会储藏所有修改过的任何东西，但是会交互式地提示哪些改动想要储藏、哪些改动需要保存在工作目录中。

```
$ git stash --patch
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 66d332e..8bb5674 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -16,6 +16,10 @@ class SimpleGit
         return `#{git_cmd} 2>&1`.chomp
```

```
        end
      end
+
+    def show(treeish = 'master')
+      command("git show #{treeish}")
+    end

 end
 test
Stash this hunk [y,n,q,a,d,/,e,?]? y

Saved working directory and index state WIP on master: 1b65b17 added the index file
```

从储藏创建一个分支

如果储藏了一些工作，将它留在那儿了一会儿，然后继续在储藏的分支上工作，在重新应用工作时可能会有问题。 如果应用尝试修改刚刚修改的文件，你会得到一个合并冲突并不得不解决它。 如果想要一个轻松的方式来再次测试储藏的改动，可以运行 `git stash branch` 创建一个新分支，检出储藏工作时所在的提交，重新在那应用工作，然后在应用成功后扔掉储藏：

```
$ git stash branch testchanges
Switched to a new branch "testchanges"
# On branch testchanges
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      modified:   index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   lib/simplegit.rb
#
Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

这是在新分支轻松恢复储藏工作并继续工作的一个很不错的途径。

清理工作目录

对于工作目录中一些工作或文件，你想做的也许不是储藏而是移除。 `git clean` 命令会帮你做这些事。

有一些通用的原因比如说为了移除由合并或外部工具生成的东西，或是为了运行一个干净的构建而移除之前构建的残留。

你需要谨慎地使用这个命令，因为它被设计为从工作目录中移除未被追踪的文件。 如果你改变主意了，你也不一定能找回来那些文件的内容。 一个更安全的选项是运行 `git stash --all` 来移除每一样东西并存放在栈中。

你可以使用 `git clean` 命令去除冗余文件或者清理工作目录。 使用 `git clean -f -d` 命令来移除工作目录中所有未追踪的文件以及空的子目录。 `-f` 意味着 强制 或 “确定移除”。

如果只是想要看看它会做什么，可以使用 `-n` 选项来运行命令，这意味着 “做一次演习然后告诉你将要 移除什么”。

```
$ git clean -d -n
Would remove test.o
Would remove tmp/
```

默认情况下，`git clean` 命令只会移除没有忽略的未跟踪文件。 任何与 `.gitiignore` 或其他忽略文件中的模式匹配的文件都不会被移除。 如果你也想要移除那些文件，例如为了做一次完全干净的构建而移除所有由构建生成的 `.o` 文件，可以给 clean 命令增加一个 `-x` 选项。

```
$ git status -s
 M lib/simplegit.rb
?? build.TMP
?? tmp/

$ git clean -n -d
Would remove build.TMP
Would remove tmp/

$ git clean -n -d -x
Would remove build.TMP
Would remove test.o
Would remove tmp/
```

如果不知道 `git clean` 命令将会做什么，在将 `-n` 改为 `-f` 来真正做之前总是先用 `-n` 来运行它做双重检查。 另一个小心处理过程的方式是使用 `-i` 或 “interactive” 标记来运行它。

这将会以交互模式运行 clean 命令。

```
$ git clean -x -i
Would remove the following items:
  build.TMP  test.o
*** Commands ***
    1: clean                2: filter by pattern    3: select by numbers    4: ask each             5:
quit
    6: help
What now>
```

这种方式下可以分别地检查每一个文件或者交互地指定删除的模式。

7.4 签署工作

签署工作

Git 虽然是密码级安全的，但它不是万无一失的。 如果你从因特网上的其他人那里拿取工作，并且想要验证提交是不是真正地来自于可信来源，Git 提供了几种通过 GPG 来签署和验证工作的方式。

GPG 介绍

首先，在开始签名之前你需要先配置 GPG 并安装个人密钥。

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
---------------------------------
pub   2048R/0A46826A 2014-06-04
uid                  Scott Chacon (Git signing key) <schacon@gmail.com>
sub   2048R/874529A9 2014-06-04
```

如果你还没有安装一个密钥，可以使用 `gpg --gen-key` 生成一个。

```
gpg --gen-key
```

一旦你有一个可以签署的私钥，可以通过设置 Git 的 `user.signingkey` 选项来签署。

```
git config --global user.signingkey 0A46826A
```

现在 Git 默认使用你的密钥来签署标签与提交。

签署标签

如果已经设置好一个 GPG 私钥，可以使用它来签署新的标签。 所有需要做的只是使用 `-s` 代替 `-a` 即可：

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'

You need a passphrase to unlock the secret key for
user: "Ben Straub <ben@straub.cc>"
2048-bit RSA key, ID 800430EB, created 2014-05-04
```

如果在那个标签上运行 `git show`，会看到你的 GPG 签名附属在后面：

```
$ git show v1.5
tag v1.5
Tagger: Ben Straub <ben@straub.cc>
```

```
Date:   Sat May 3 20:29:41 2014 -0700

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQEcBAABAgAGBQJTZbQlAAoJEF0+sviABDDrZbQH/09PfE51KPVPlanr6q1v4/Ut
LQxfojUWiLQdg2ESJItkcuweYg+kc3HCyFejeDIBw9dpXt00rY26p05qrpnG+85b
hM1/PswpPLuBSr+oCIDj5GMC2r2iEKsfv2fJbNW8iWAXVLoWZRF8B0MfqX/YTMbm
ecorc4iXzQu7tupRihslbNkfvfciMnSDeSvzCpWAHl7h8Wj6hhqePmLm9lAYqnKp
8S5B/1SSQuEAjRZgI4IexpZoeKGVDptPHxLLS38fozsyi0QyDyzEgJxcJQVMXxVi
RUysgqjcpT8+iQM1PblGfHR4XAhuOqN5Fx06PSaFZhqvWFezJ28/CLyX5q+oIVk=
=EFTF
-----END PGP SIGNATURE-----

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

验证标签

要验证一个签署的标签，可以运行 `git tag -v [tag-name]`。 这个命令使用 GPG 来验证签名。 为了验证能正常工作，签署者的公钥需要在你的钥匙链中。

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
gpg:                 aka "[jpeg image of size 1513]"
Primary key fingerprint: 3565 2A26 2040 E066 C9A7  4A7D C0C6 D9A4 F311 9B9A
```

如果没有签署者的公钥，那么你将会得到类似下面的东西：

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

签署提交

在最新版本的 Git 中（v1.7.9 及以上），也可以签署个人提交。 如果相对于标签而言你对直接签署到提交更感兴趣的话，所有要做的只是增加一个 `-S` 到 `git commit` 命令。

```
$ git commit -a -S -m 'signed commit'

You need a passphrase to unlock the secret key for
```

```
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04

[master 5c3386c] signed commit
 4 files changed, 4 insertions(+), 24 deletions(-)
 rewrite Rakefile (100%)
 create mode 100644 lib/git.rb
```

`git log` 也有一个 `--show-signature` 选项来查看及验证这些签名。

```
$ git log --show-signature -1
commit 5c3386cf54bba0a33a32da706aa52bc0155503c2
gpg: Signature made Wed Jun  4 19:49:17 2014 PDT using RSA key ID 0A46826A
gpg: Good signature from "Scott Chacon (Git signing key) <schacon@gmail.com>"
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Jun 4 19:49:17 2014 -0700

    signed commit
```

另外，也可以配置 `git log` 来验证任何找到的签名并将它们以 `%G?` 格式列在输出中。

```
$ git log --pretty="format:%h %G? %aN  %s"

5c3386c G Scott Chacon  signed commit
ca82a6d N Scott Chacon  changed the version number
085bb3b N Scott Chacon  removed unnecessary test code
a11bef0 N Scott Chacon  first commit
```

这里我们可以看到只有最后一次提交是签署并有效的，而之前的提交都不是。

在 Git 1.8.3 及以后的版本中，“git merge” 与 “git pull” 可以使用 `--verify-signatures` 选项来检查并拒绝没有携带可信 GPG 签名的提交。

如果使用这个选项来合并一个包含未签名或有效的提交的分支时，合并不会生效。

```
$ git merge --verify-signatures non-verify
fatal: Commit ab06180 does not have a GPG signature.
```

如果合并包含的只有有效的签名的提交，合并命令会提示所有的签名它已经检查过了然后会继续向前。

```
$ git merge --verify-signatures signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key) <schacon@gmail.com>
Updating 5c3386c..13ad65e
Fast-forward
 README | 2 ++
 1 file changed, 2 insertions(+)
```

也可以给 `git merge` 命令附加 `-S` 选项来签署自己生成的合并提交。 下面的例子演示了验证将要合

并的分支的每一个提交都是签名的并且签署最后生成的合并提交。

```
$ git merge --verify-signatures -S  signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key) <schacon@gmail.com>

You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04

Merge made by the 'recursive' strategy.
 README | 2 ++
 1 file changed, 2 insertions(+)
```

每个人必须签署

签署标签与提交很棒，但是如果决定在正常的工作流程中使用它，你必须确保团队中的每一个人都理解如何这样做。 如果没有，你将会花费大量时间帮助其他人找出并用签名的版本重写提交。 在采用签署成为标准工作流程的一部分前，确保你完全理解 GPG 及签署带来的好处。

7.5 搜索

搜索

无论仓库里的代码量有多少，你经常需要查找一个函数是在哪里调用或者定义的，或者一个方法的变更历史。 Git 提供了两个有用的工具来快速地从它的数据库中浏览代码和提交。 我们来简单的看一下。

Git Grep

Git 提供了一个 `grep` 命令，你可以很方便地从提交历史或者工作目录中查找一个字符串或者正则表达式。 我们用 Git 本身源代码的查找作为例子。

默认情况下 Git 会查找你工作目录的文件。 你可以传入 `-n` 参数来输出 Git 所找到的匹配行行号。

```
$ git grep -n gmtime_r
compat/gmtime.c:3:#undef gmtime_r
compat/gmtime.c:8:      return git_gmtime_r(timep, &result);
compat/gmtime.c:11:struct tm *git_gmtime_r(const time_t *timep, struct tm *result)
compat/gmtime.c:16:     ret = gmtime_r(timep, result);
compat/mingw.c:606:struct tm *gmtime_r(const time_t *timep, struct tm *result)
compat/mingw.h:162:struct tm *gmtime_r(const time_t *timep, struct tm *result);
date.c:429:             if (gmtime_r(&now, &now_tm))
date.c:492:             if (gmtime_r(&time, tm)) {
git-compat-util.h:721:struct tm *git_gmtime_r(const time_t *, struct tm *);
git-compat-util.h:723:#define gmtime_r git_gmtime_r
```

`grep` 命令有一些有趣的选项。

例如，你可以使用 `--count` 选项来使 Git 输出概述的信息，仅仅包括哪些文件包含匹配以及每个文件包含了多少个匹配。

```
$ git grep --count gmtime_r
compat/gmtime.c:4
compat/mingw.c:1
compat/mingw.h:1
date.c:2
git-compat-util.h:2
```

如果你想看匹配的行是属于哪一个方法或者函数，你可以传入 `-p` 选项：

```
$ git grep -p gmtime_r *.c
date.c=static int match_multi_number(unsigned long num, char c, const char *date, char *end, struct tm
*tm)
date.c:         if (gmtime_r(&now, &now_tm))
date.c=static int match_digit(const char *date, struct tm *tm, int *offset, int *tm_gmt)
date.c:         if (gmtime_r(&time, tm)) {
```

在这里我们可以看到在 date.c 文件中有 `match_multi_number` 和 `match_digit` 两个函数调用了 `gmtime_r`。

你还可以使用 `--and` 标志来查看复杂的字符串组合，也就是在同一行同时包含多个匹配。 比如，我们要查看在旧版本 1.8.0 的 Git 代码库中定义了常量名包含 “LINK” 或者 “BUF_MAX” 这两个字符串所在的行。

这里我们也用到了 `--break` 和 `--heading` 选项来使输出更加容易阅读。

```
$ git grep --break --heading \
    -n -e '#define' --and \( -e LINK -e BUF_MAX \) v1.8.0
v1.8.0:builtin/index-pack.c
62:#define FLAG_LINK (1u<<20)

v1.8.0:cache.h
73:#define S_IFGITLINK  0160000
74:#define S_ISGITLINK(m)       (((m) & S_IFMT) == S_IFGITLINK)

v1.8.0:environment.c
54:#define OBJECT_CREATION_MODE OBJECT_CREATION_USES_HARDLINKS

v1.8.0:strbuf.c
326:#define STRBUF_MAXLINK (2*PATH_MAX)

v1.8.0:symlinks.c
53:#define FL_SYMLINK  (1 << 2)

v1.8.0:zlib.c
30:/* #define ZLIB_BUF_MAX ((uInt)-1) */
31:#define ZLIB_BUF_MAX ((uInt) 1024 * 1024 * 1024) /* 1GB */
```

相比于一些常用的搜索命令比如 `grep` 和 `ack`，`git grep` 命令有一些的优点。 第一就是速度非常快，第二是你不仅仅可以可以搜索工作目录，还可以搜索任意的 Git 树。 在上一个例子中，我们在一个旧版本的 Git 源代码中查找，而不是当前检出的版本。

Git 日志搜索

或许你不想知道某一项在 哪里 ，而是想知道是什么 时候 存在或者引入的。 `git log` 命令有许多强大的工具可以通过提交信息甚至是 diff 的内容来找到某个特定的提交。

例如，如果我们想找到 `ZLIB_BUF_MAX` 常量是什么时候引入的，我们可以使用 `-S` 选项来显示新增和删除该字符串的提交。

```
$ git log -SZLIB_BUF_MAX --oneline
e01503b zlib: allow feeding more than 4GB in one go
ef49a7a zlib: zlib can only process 4GB at a time
```

如果我们查看这些提交的 diff，我们可以看到在 `ef49a7a` 这个提交引入了常量，并且在 `e01503b` 这个提交中被修改了。

如果你希望得到更精确的结果，你可以使用 `-G` 选项来使用正则表达式搜索。

行日志搜索
行日志搜索是另一个相当高级并且有用的日志搜索功能。 这是一个最近新增的不太知名的功能，但却是十分有用。 在 `git log` 后加上 `-L` 选项即可调用，它可以展示代码中一行或者一个函数的历史。

例如，假设我们想查看 `zlib.c` 文件中 `git_deflate_bound` 函数的每一次变更，我们可以执行 `git log -L :git_deflate_bound:zlib.c`。 Git 会尝试找出这个函数的范围，然后查找历史记录，并且显示从函数创建之后一系列变更对应的补丁。

```
$ git log -L :git_deflate_bound:zlib.c
commit ef49a7a0126d64359c974b4b3b71d7ad42ee3bca
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:52:15 2011 -0700

    zlib: zlib can only process 4GB at a time

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -85,5 +130,5 @@
-unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+unsigned long git_deflate_bound(git_zstream *strm, unsigned long size)
 {
-       return deflateBound(strm, size);
+       return deflateBound(&strm->z, size);
 }


commit 225a6f1068f71723a910e8565db4e252b3ca21fa
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:18:17 2011 -0700

    zlib: wrap deflateBound() too

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -81,0 +85,5 @@
+unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+{
+       return deflateBound(strm, size);
+}
+
```

如果 Git 无法计算出如何匹配你代码中的函数或者方法，你可以提供一个正则表达式。 例如，这个命令和上面的是等同的：`git log -L '/unsigned long git_deflate_bound/',/^}/:zlib.c`。 你也可以提供单行或者一个范围的行号来获得相同的输出。

7.6 重写历史

重写历史

许多时候，在使用 Git 时，可能会因为某些原因想要修正提交历史。 Git 很棒的一点是它允许你在最后时刻做决定。 你可以在将暂存区内容提交前决定哪些文件进入提交，可以通过 stash 命令来决定不与某些内容工作，也可以重写已经发生的提交就像它们以另一种方式发生的一样。 这可能涉及改变提交的顺序，改变提交中的信息或修改文件，将提交压缩或是拆分，或完全地移除提交 - 在将你的工作成果与他人共享之前。

在本节中，你可以学到如何完成这些非常有用的工作，这样在与他人分享你的工作成果时你的提交历史将如你所愿地展示出来。

修改最后一次提交

修改你最近一次提交可能是所有修改历史提交的操作中最常见的一个。 对于你的最近一次提交，你往往想做两件事情：修改提交信息，或者修改你添加、修改和移除的文件的快照。

如果，你只是想修改最近一次提交的提交信息，那么很简单：

```
$ git commit --amend
```

这会把你带入文本编辑器，里面包含了你最近一条提交信息，供你修改。 当保存并关闭编辑器后，编辑器将会用你输入的内容替换最近一条提交信息。

如果你已经完成提交，又因为之前提交时忘记添加一个新创建的文件，想通过添加或修改文件来更改提交的快照，也可以通过类似的操作来完成。 通过修改文件然后运行 `git add` 或 `git rm` 一个已追踪的文件，随后运行 `git commit --amend` 拿走当前的暂存区域并使其做为新提交的快照。

使用这个技巧的时候需要小心，因为修正会改变提交的 SHA-1 校验和。 它类似于一个小的变基 - 如果已经推送了最后一次提交就不要修正它。

修改多个提交信息

为了修改在提交历史中较远的提交，必须使用更复杂的工具。 Git 没有一个改变历史工具，但是可以使用变基工具来变基一系列提交，基于它们原来的 HEAD 而不是将其移动到另一个新的上面。 通过交互式变基工具，可以在任何想要修改的提交后停止，然后修改信息、添加文件或做任何想做的事情。 可以通过给 `git rebase` 增加 `-i` 选项来交互式地运行变基。 必须指定想要重写多久远的历史，这可以通过告诉命令将要变基到的提交来做到。

例如，如果想要修改最近三次提交信息，或者那组提交中的任意一个提交信息，将想要修改的最近一次提交的父提交作为参数传递给 `git rebase -i` 命令，即 `HEAD~2^` 或 `HEAD~3`。 记住 `~3` 可能比较容易，因为你正尝试修改最后三次提交；但是注意实际上指定了以前的四次提交，即想要修改提交的父提

交：

```
$ git rebase -i HEAD~3
```

再次记住这是一个变基命令 - 在 `HEAD~3..HEAD` 范围内的每一个提交都会被重写，无论你是否修改信息。 不要涉及任何已经推送到中央服务器的提交 - 这样做会产生一次变更的两个版本，因而使他人困惑。

运行这个命令会在文本编辑器上给你一个提交的列表，看起来像下面这样：

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

需要重点注意的是相对于正常使用的 `log` 命令，这些提交显示的顺序是相反的。 运行一次 log 命令，会看到类似这样的东西：

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d added cat-file
310154e updated README formatting and added blame
f7f3f6d changed my name a bit
```

注意其中的反序显示。 交互式变基给你一个它将会运行的脚本。 它将会从你在命令行中指定的提交（`HEAD~3`）开始，从上到下的依次重演每一个提交引入的修改。 它将最旧的而不是最新的列在上面，因为那会是第一个将要重演的。

你需要修改脚本来让它停留在你想修改的变更上。 要达到这个目的，你只要将你想修改的每一次提交前面的 ‘pick’ 改为 ‘edit’ 。 例如，只想修改第三次提交信息，可以像下面这样修改文件：

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

当保存并退出编辑器时，Git 将你带回到列表中的最后一次提交，把你送回命令行并提示以下信息：

```
$ git rebase -i HEAD~3
Stopped at f7f3f6d... changed my name a bit
You can amend the commit now, with

       git commit --amend

Once you're satisfied with your changes, run

       git rebase --continue
```

这些指令准确地告诉你该做什么。 输入

```
$ git commit --amend
```

修改提交信息，然后退出编辑器。 然后，运行

```
$ git rebase --continue
```

这个命令将会自动地应用另外两个提交，然后就完成了。 如果需要将不止一处的 pick 改为 edit，需要在每一个修改为 edit 的提交上重复这些步骤。 每一次，Git 将会停止，让你修正提交，然后继续直到完成。

重新排序提交

也可以使用交互式变基来重新排序或完全移除提交。 如果想要移除 “added cat-file” 提交然后修改另外两个提交引入的顺序，可以将变基脚本从这样：

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

改为这样：

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

当保存并退出编辑器时，Git 将你的分支带回这些提交的父提交，应用 `310154e` 然后应用 `f7f3f6d`，最后停止。 事实修改了那些提交的顺序并完全地移除了 “added cat-file” 提交。

压缩提交

通过交互式变基工具，也可以将一连串提交压缩成一个单独的提交。 在变基信息中脚本给出了有用的指

令：

```
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

如果，指定 “squash” 而不是 “pick” 或 “edit” ，Git 将应用两者的修改并合并提交信息在一起。 所以，如果想要这三次提交变为一个提交，可以这样修改脚本：

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

当保存并退出编辑器时，Git 应用所有的三次修改然后将你放到编辑器中来合并三次提交信息：

```
# This is a combination of 3 commits.
# The first commit's message is:
changed my name a bit

# This is the 2nd commit message:

updated README formatting and added blame

# This is the 3rd commit message:

added cat-file
```

当你保存之后，你就拥有了一个包含前三次提交的全部变更的提交。

拆分提交

拆分一个提交会撤消这个提交，然后多次地部分地暂存与提交直到完成你所需次数的提交。 例如，假设想要拆分三次提交的中间那次提交。 想要将它拆分为两次提交：第一个 “updated README formatting” ，第二个 “added blame” 来代替原来的 “updated README formatting and added blame” 。 可以通过修改 `rebase -i` 的脚本来做到这点，将要拆分的提交的指令修改为 “edit” ：

```
pick f7f3f6d changed my name a bit
```

```
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

然后，当脚本将你进入到命令行时，重置那个提交，拿到被重置的修改，从中创建几次提交。 当保存并退出编辑器时，Git 带你到列表中第一个提交的父提交，应用第一个提交（ `f7f3f6d` ），应用第二个提交（ `310154e` ），然后让你进入命令行。 那里，可以通过 `git reset HEAD^` 做一次针对那个提交的混合重置，实际上将会撤消那次提交并将修改的文件未暂存。 现在可以暂存并提交文件直到有几个提交，然后当完成时运行 `git rebase --continue` ：

```
$ git reset HEAD^
$ git add README
$ git commit -m 'updated README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'added blame'
$ git rebase --continue
```

Git 在脚本中应用最后一次提交（ `a5f4a0d` ），历史记录看起来像这样：

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

再一次，这些改动了所有在列表中的提交的 SHA-1 校验和，所以要确保列表中的提交还没有推送到共享仓库中。

核武器级选项：filter-branch

有另一个历史改写的选项，如果想要通过脚本的方式改写大量提交的话可以使用它 - 例如，全局修改你的邮箱地址或从每一个提交中移除一个文件。 这个命令是 `filter-branch` ，它可以改写历史中大量的提交，除非你的项目还没有公开并且其他人没有基于要改写的工作的提交做的工作，你不应当使用它。 然而，它可以很有用。 你将会学习到几个常用的用途，这样就得到了它适合使用地方的想法。

从每一个提交移除一个文件

这经常发生。 有人粗心地通过 `git add .` 提交了一个巨大的二进制文件，你想要从所有地方删除它。 可能偶然地提交了一个包括一个密码的文件，然而你想要开源项目。 `filter-branch` 是一个可能会用来擦洗整个提交历史的工具。 为了从整个提交历史中移除一个叫做 passwords.txt 的文件，可以使用 `--tree-filter` 选项给 `filter-branch` ：

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

`--tree-filter` 选项在检出项目的每一个提交后运行指定的命令然后重新提交结果。 在本例中，你从

每一个快照中移除了一个叫作 passwords.txt 的文件，无论它是否存在。 如果想要移除所有偶然提交的编辑器备份文件，可以运行类似 `git filter-branch --tree-filter 'rm -f *~' HEAD` 的命令。

最后将可以看到 Git 重写树与提交然后移动分支指针。 通常一个好的想法是在一个测试分支中做这件事，然后当你决定最终结果是真正想要的，可以硬重置 master 分支。 为了让 `filter-branch` 在所有分支上运行，可以给命令传递 `--all` 选项。

使一个子目录做为新的根目录
假设已经从另一个源代码控制系统中导入，并且有几个没意义的子目录（trunk、tags 等等）。 如果想要让 `trunk` 子目录作为每一个提交的新的项目根目录，`filter-branch` 也可以帮助你那么做：

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

现在新项目根目录是 `trunk` 子目录了。 Git 会自动移除所有不影响子目录的提交。

全局修改邮箱地址
另一个常见的情形是在你开始工作时忘记运行 `git config` 来设置你的名字与邮箱地址，或者你想要开源一个项目并且修改所有你的工作邮箱地址为你的个人邮箱地址。 任何情形下，你也可以通过 `filter-branch` 来一次性修改多个提交中的邮箱地址。 需要小心的是只修改你自己的邮箱地址，所以你使用 `--commit-filter`：

```
$ git filter-branch --commit-filter '
        if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
        then
                GIT_AUTHOR_NAME="Scott Chacon";
                GIT_AUTHOR_EMAIL="schacon@example.com";
                git commit-tree "$@";
        else
                git commit-tree "$@";
        fi' HEAD
```

这会遍历并重写每一个提交来包含你的新邮箱地址。 因为提交包含了它们父提交的 SHA-1 校验和，这个命令会修改你的历史中的每一个提交的 SHA-1 校验和，而不仅仅只是那些匹配邮箱地址的提交。

7.7 重置揭密

重置揭密

在继续了解更专业的工具前，我们先讨论一下 `reset` 与 `checkout`。 在你初次遇到的 Git 命令中，这两个是最让人困惑的。 它们能做很多事情，所以看起来我们很难真正地理解并恰当地运用它们。 针对这一点，我们先来做一个简单的比喻。

三棵树

理解 `reset` 和 `checkout` 的最简方法，就是以 Git 的思维框架（将其作为内容管理器）来管理三棵不同的树。 “树” 在我们这里的实际意思是 “文件的集合” ，而不是指特定的数据结构。 （在某些情况下索引看起来并不像一棵树，不过我们现在的目的是用简单的方式思考它。）

Git 作为一个系统，是以它的一般操作来管理并操纵这三棵树的：

树	用途

HEAD 上一次提交的快照，下一次提交的父结点

|
|

Index

|

预期的下一次提交的快照

|
|

Working Directory

|

沙盒

|

HEAD

HEAD 是当前分支引用的指针，它总是指向该分支上的最后一次提交。 这表示 HEAD 将是下一次提交的父结点。 通常，理解 HEAD 的最简方式，就是将它看做 你的上一次提交 的快照。

其实，查看快照的样子很容易。 下例就显示了 HEAD 快照实际的目录列表，以及其中每个文件的 SHA-1

校验和：

```
$ git cat-file -p HEAD
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
author Scott Chacon  1301511835 -0700
committer Scott Chacon  1301511835 -0700

initial commit

$ git ls-tree -r HEAD
100644 blob a906cb2a4a904a152...   README
100644 blob 8f94139338f9404f2...   Rakefile
040000 tree 99f1a6d12cb4b6f19...   lib
```

`cat-file` 与 `ls-tree` 是底层命令，它们一般用于底层工作，在日常工作中并不使用。不过它们能帮助我们了解到底发生了什么。

索引
索引是你的 预期的下一次提交。 我们也会将这个概念引用为 Git 的 “暂存区域” ，这就是当你运行 `git commit` 时 Git 看起来的样子。

Git 将上一次检出到工作目录中的所有文件填充到索引区，它们看起来就像最初被检出时的样子。 之后你会将其中一些文件替换为新版本，接着通过 `git commit` 将它们转换为树来用作新的提交。

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0       README
100644 8f94139338f9404f26296befa88755fc2598c289 0       Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0       lib/simplegit.rb
```

再说一次，我们在这里又用到了 `ls-files` 这个幕后的命令，它会显示出索引当前的样子。

确切来说，索引并非技术上的树结构，它其实是以扁平的清单实现的。不过对我们而言，把它当做树就够了。

工作目录
最后，你就有了自己的工作目录。 另外两棵树以一种高效但并不直观的方式，将它们的内容存储在 `.git` 文件夹中。 工作目录会将它们解包为实际的文件以便编辑。 你可以把工作目录当做 沙盒。在你将修改提交到暂存区并记录到历史之前，可以随意更改。

```
$ tree
.
├── README
├── Rakefile
└── lib
    └── simplegit.rb

1 directory, 3 files
```

工作流程

Git 主要的目的是通过操纵这三棵树来以更加连续的状态记录项目的快照。

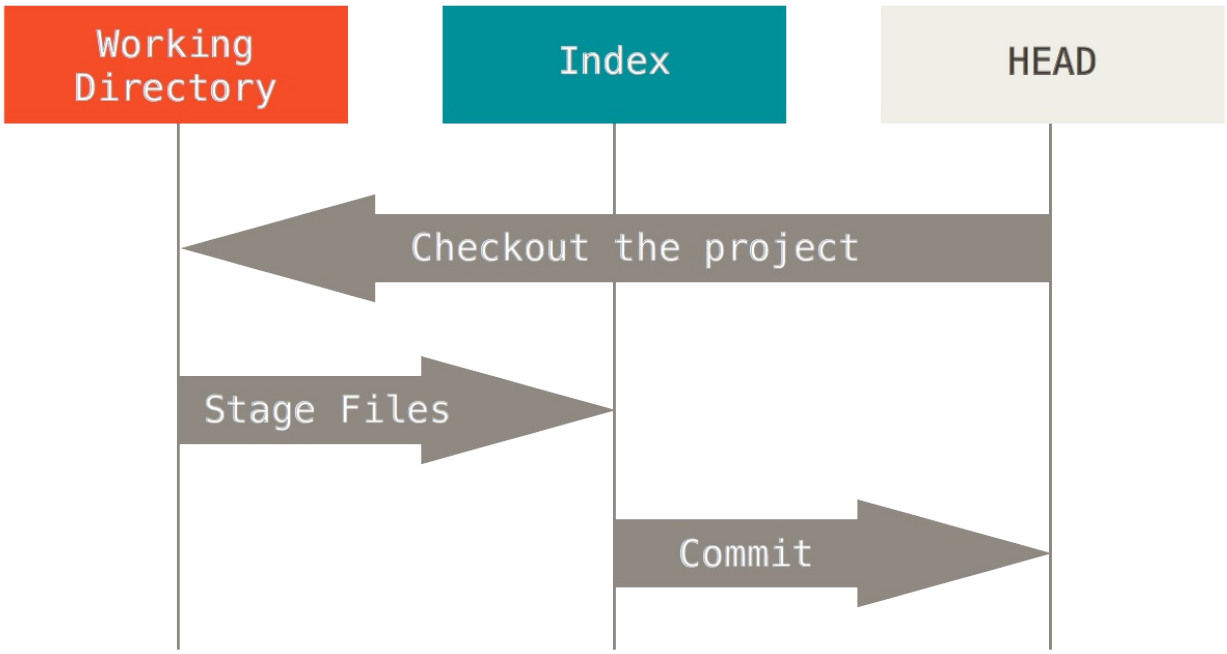


Figure 7-2.

让我们来可视化这个过程：假设我们进入到一个新目录，其中有一个文件。 我们称其为该文件的 v1版本，将它标记为蓝色。 现在运行 `git init`，这会创建一个 Git 仓库，其中的 HEAD 引用指向未创建的分支（ `master` 还不存在）。

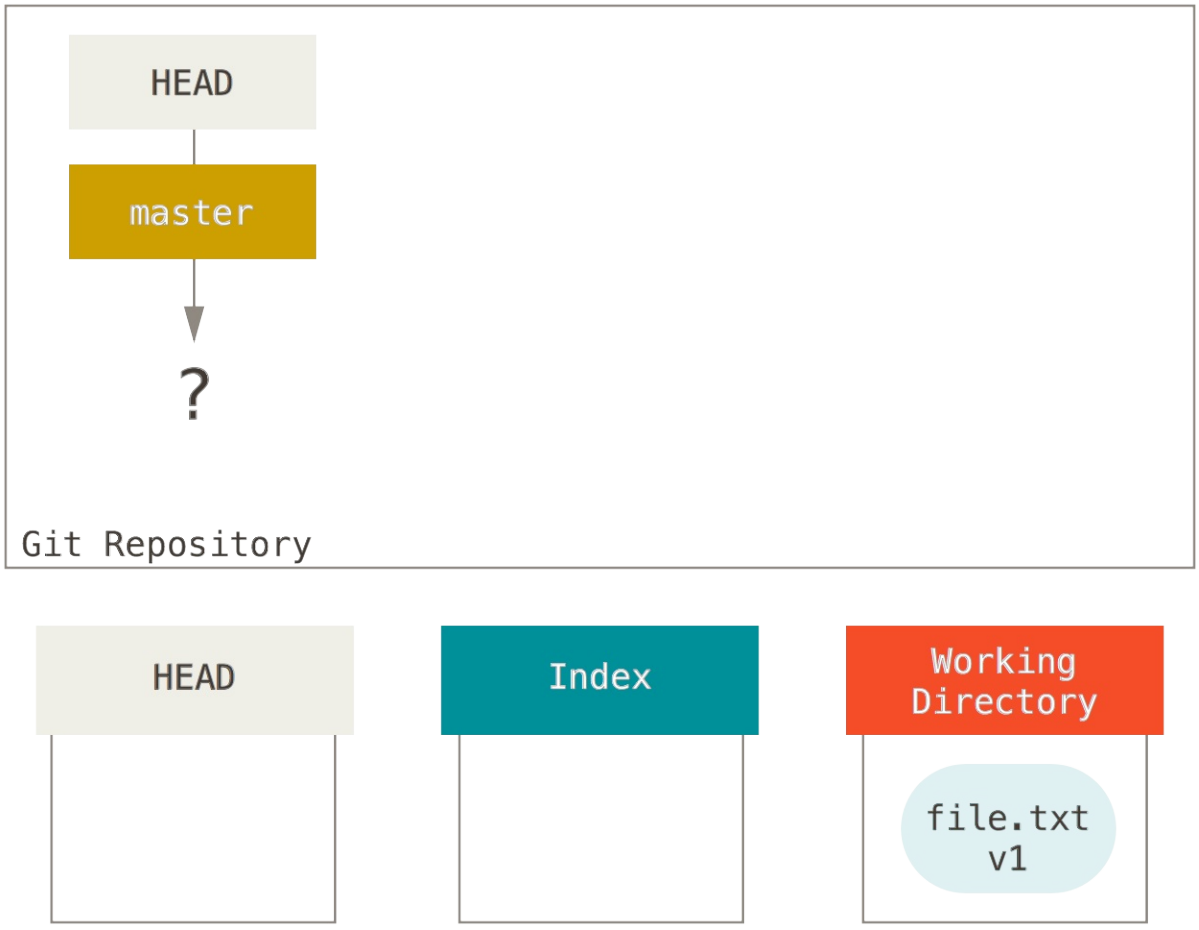


Figure 7-3.

此时，只有工作目录有内容。

现在我们想要提交这个文件，所以用 `git add` 来获取工作目录中的内容，并将其复制到索引中。

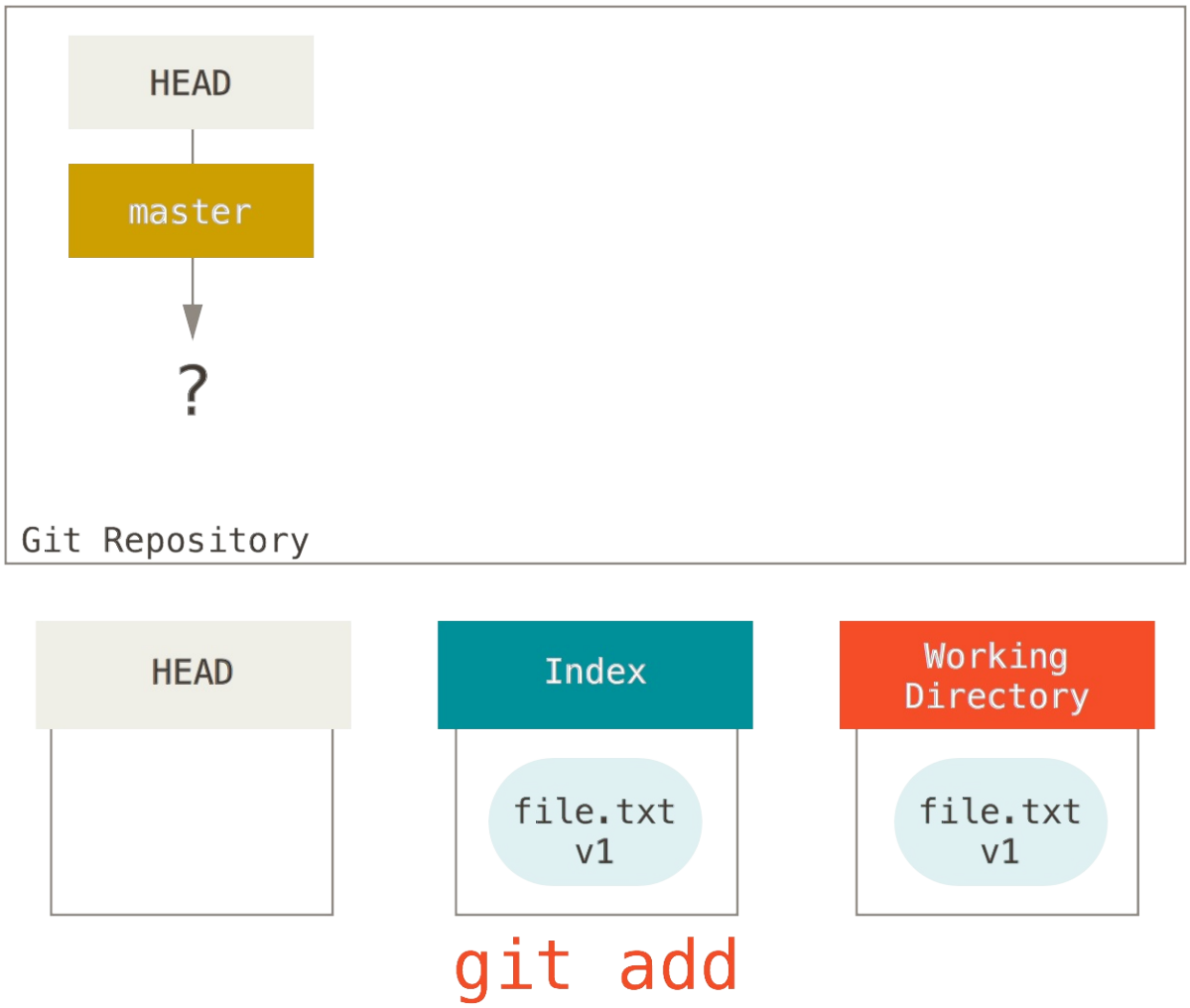


Figure 7-4.

接着运行 `git commit`，它首先会移除索引中的内容并将它保存为一个永久的快照，然后创建一个指向该快照的提交对象，最后更新 `master` 来指向本次提交。

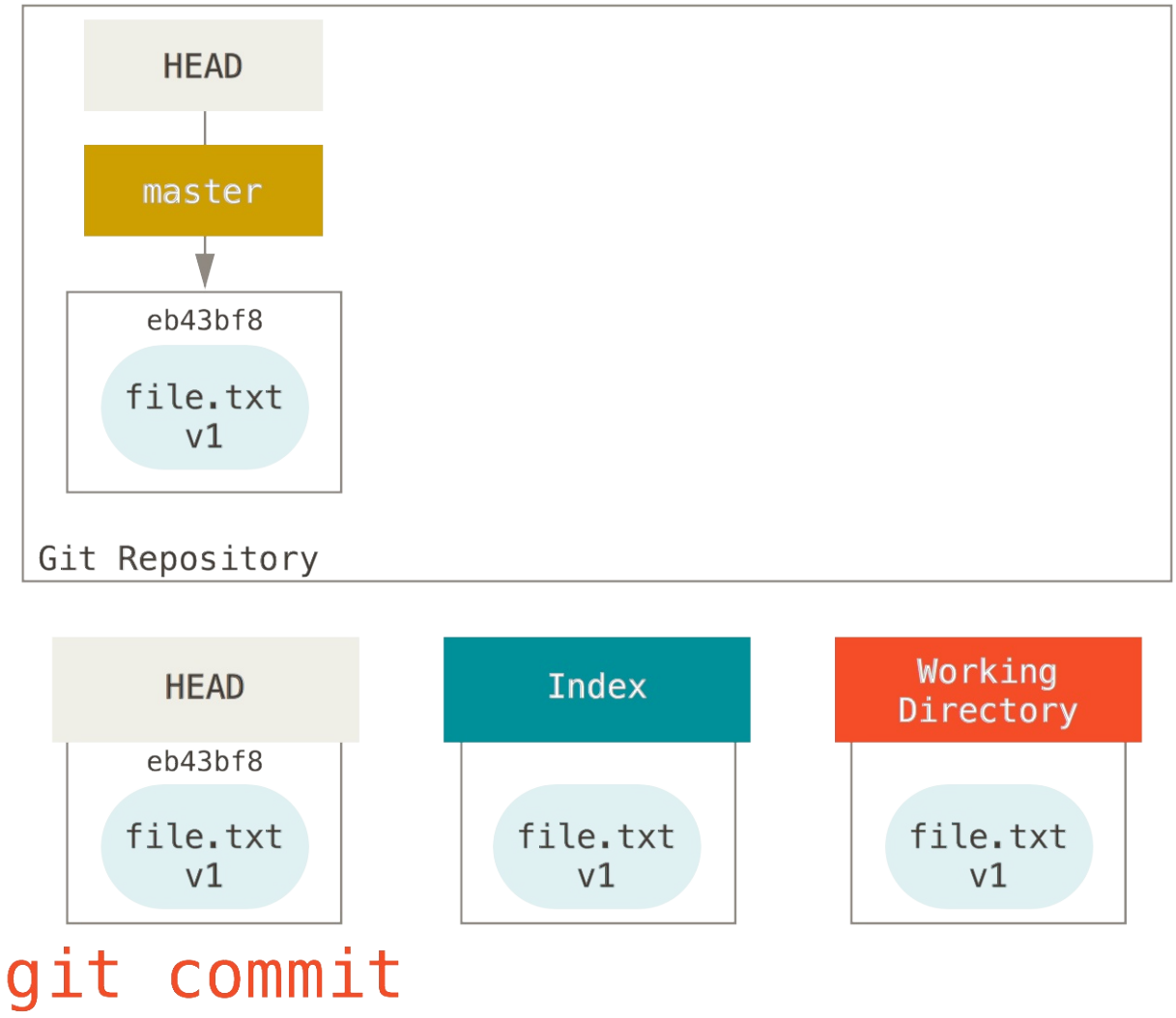


Figure 7-5.

此时如果我们运行 `git status`，会发现没有任何改动，因为现在三棵树完全相同。

现在我们想要对文件进行修改然后提交它。 我们将会经历同样的过程；首先在工作目录中修改文件。 我们称其为该文件的 v2 版本，并将它标记为红色。

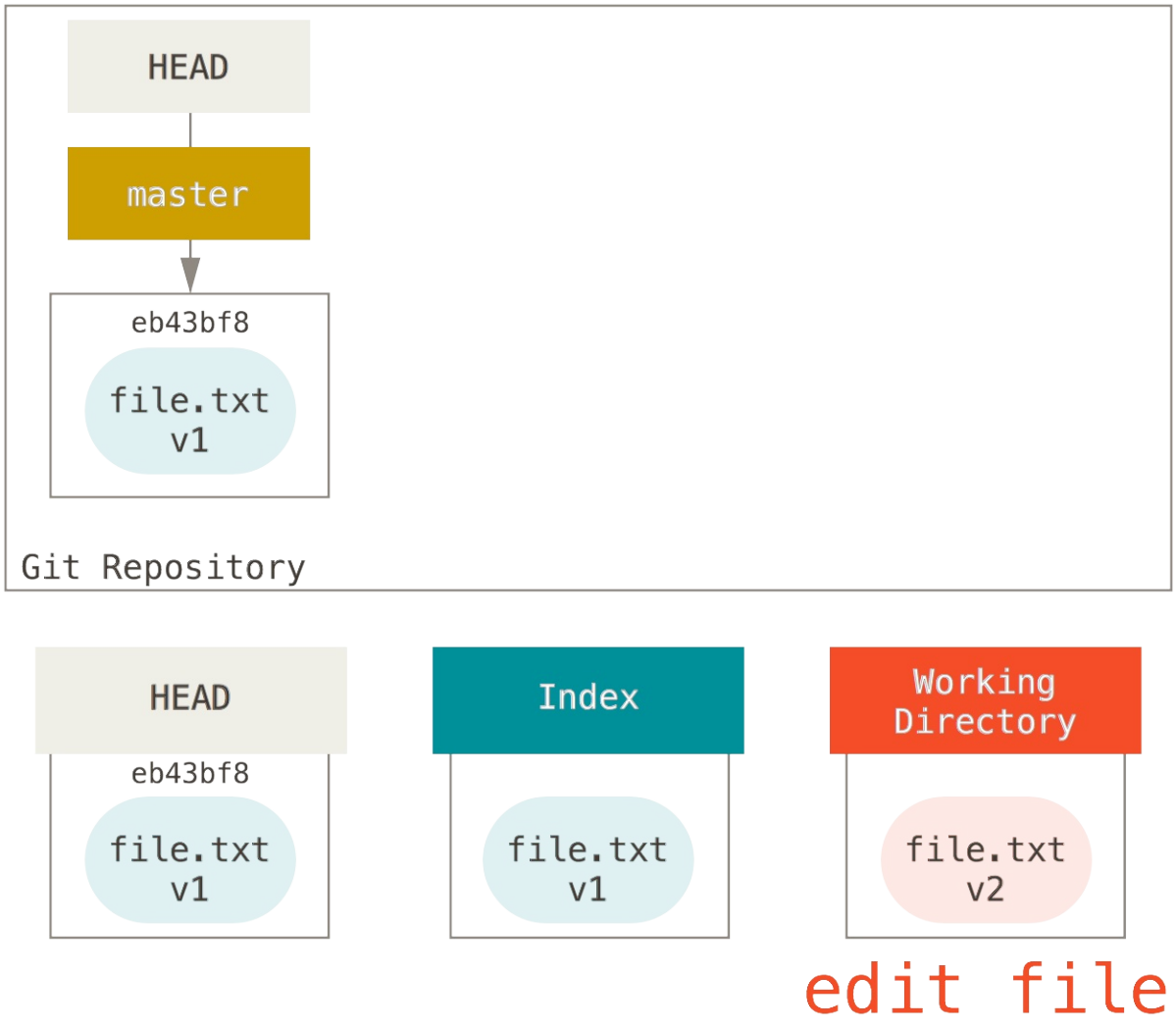


Figure 7-6.

如果现在运行 `git status`，我们会看到文件显示在 “Changes not staged for commit,” 下面并被标记为红色，因为该条目在索引与工作目录之间存在不同。 接着我们运行 `git add` 来将它暂存到索引中。

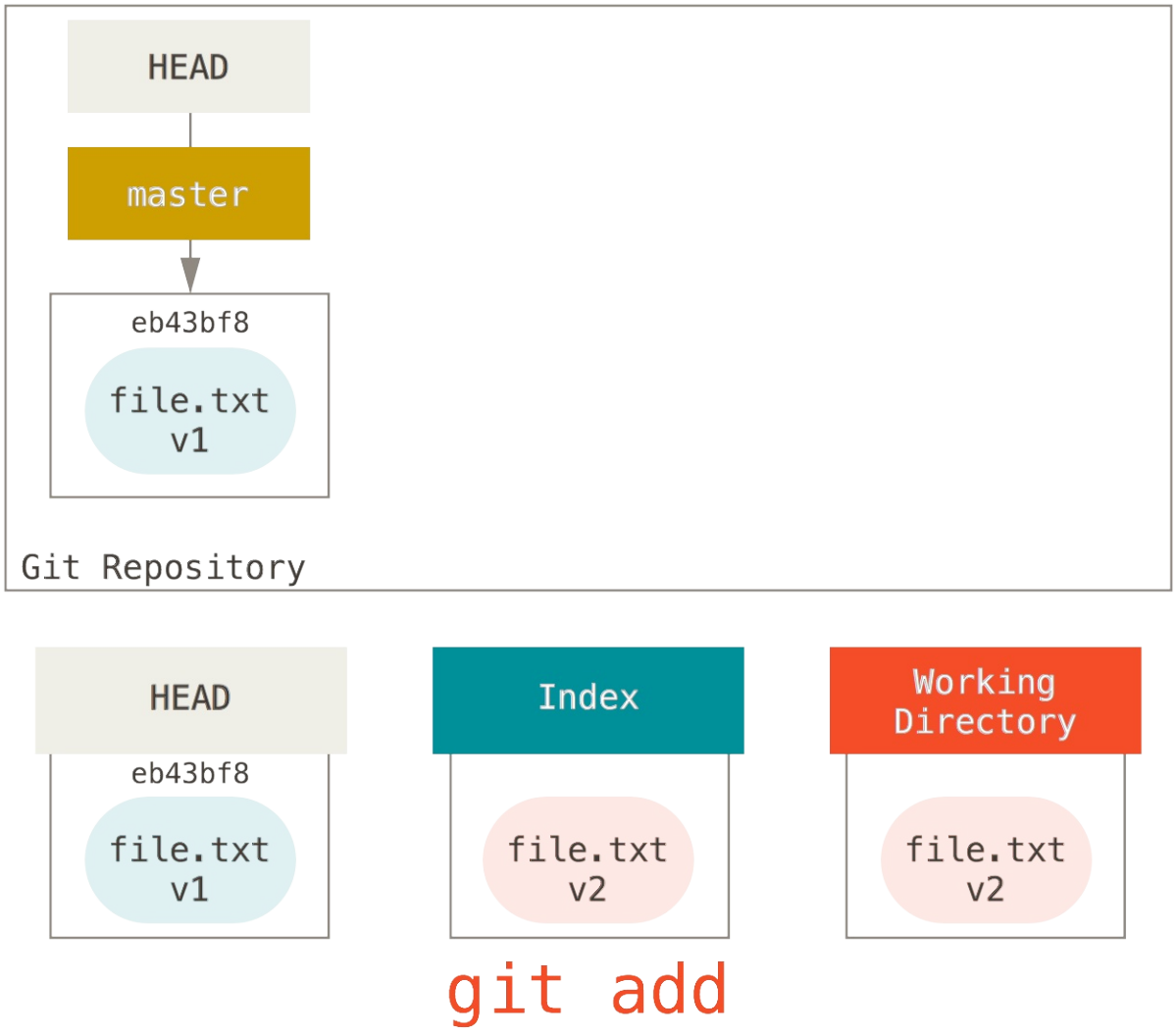


Figure 7-7.

此时，由于索引和 HEAD 不同，若运行 `git status` 的话就会看到 “Changes to be committed” 下的该文件变为绿色 ——也就是说，现在预期的下一次提交与上一次提交不同。 最后，我们运行 `git commit` 来完成提交。

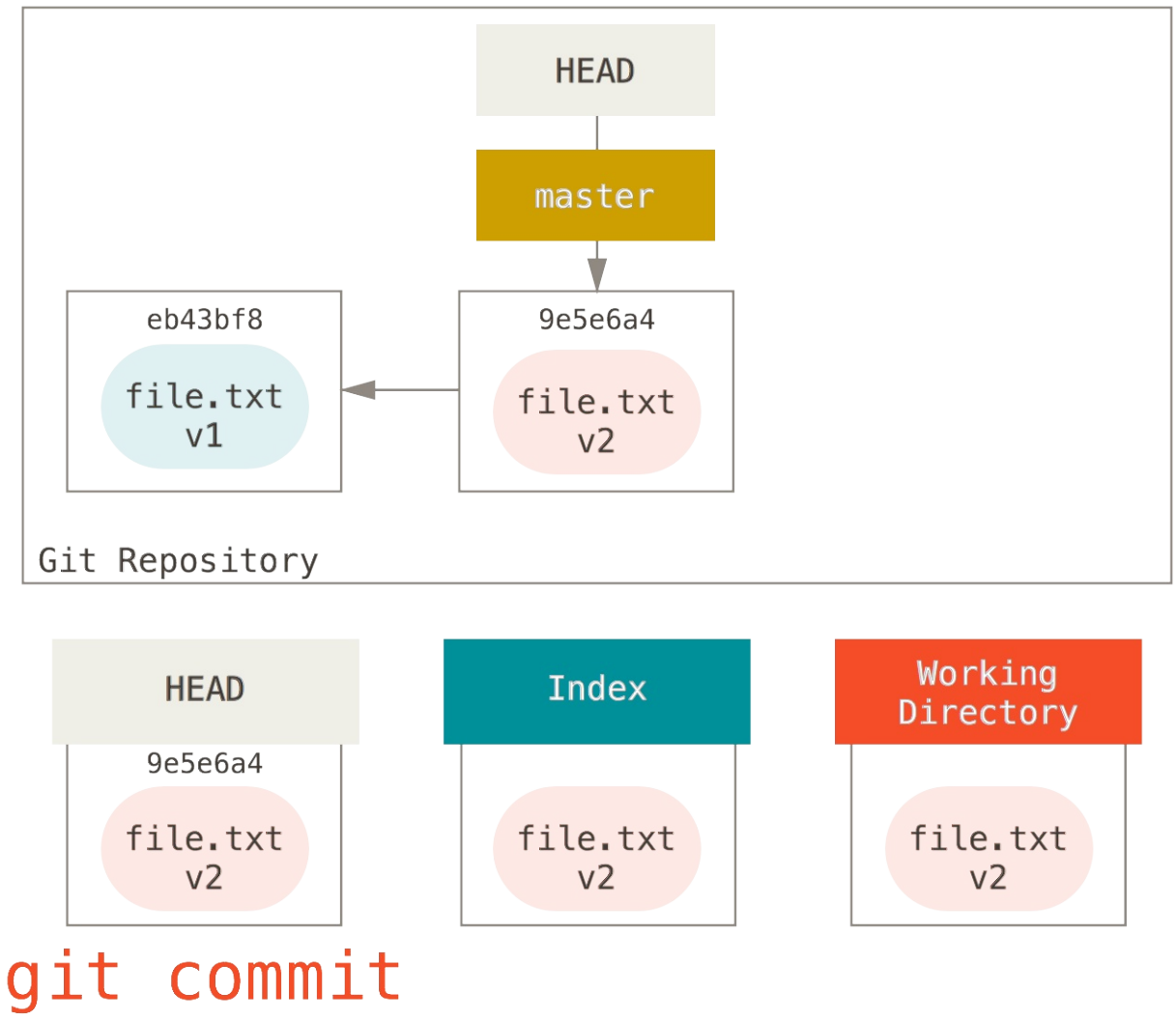


Figure 7-8.

现在运行 `git status` 会没有输出，因为三棵树又变得相同了。

切换分支或克隆的过程也类似。 当检出一个分支时，它会修改 HEAD 指向新的分支引用，将 索引 填充为该次提交的快照，然后将 索引 的内容复制到 工作目录 中。

重置的作用

在以下情景中观察 `reset` 命令会更有意义。

为了演示这些例子，假设我们再次修改了 `file.txt` 文件并第三次提交它。 现在的历史看起来是这样的：

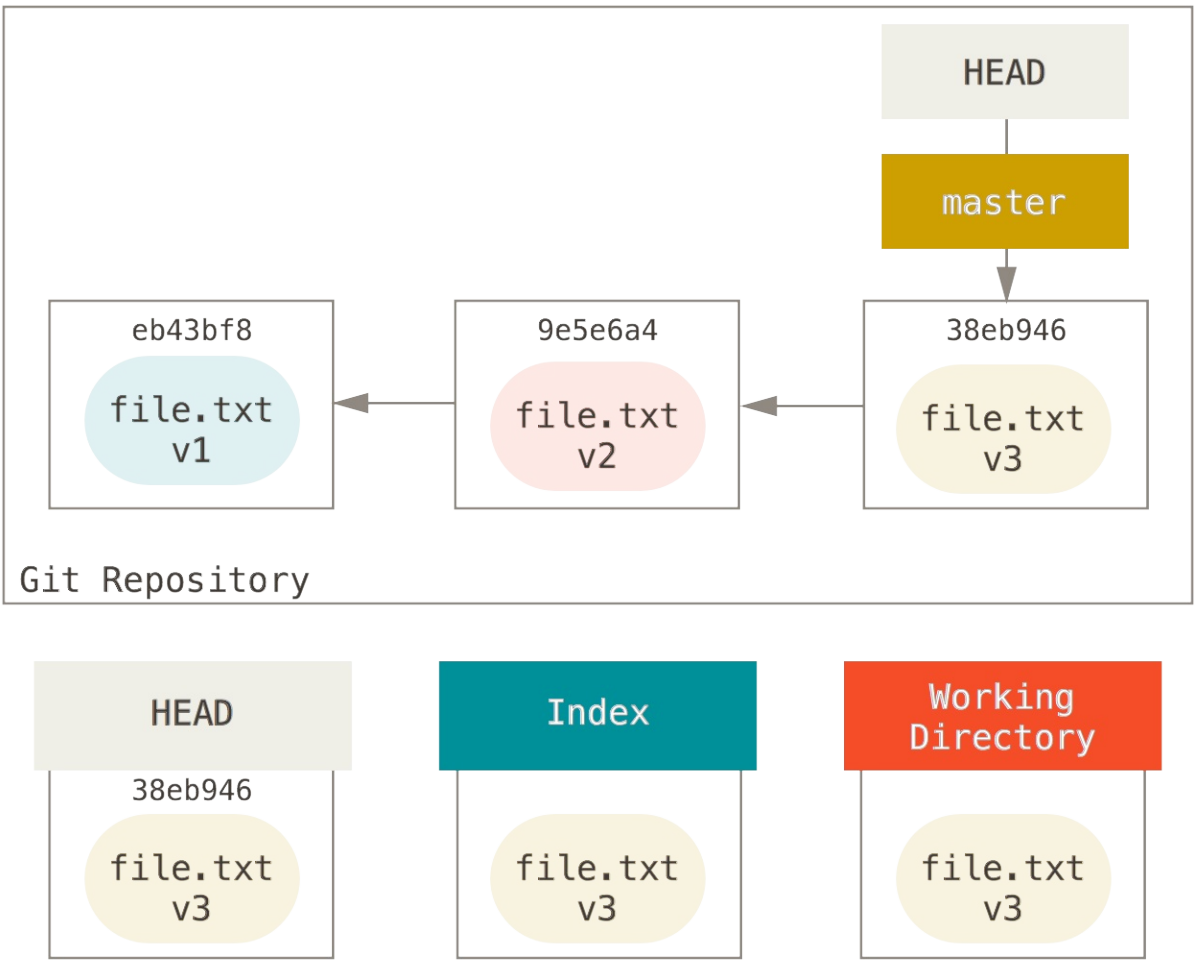


Figure 7-9.

让我们跟着 `reset` 看看它都做了什么。 它以一种简单可预见的方式直接操纵这三棵树。 它做了三个基本操作。

第 1 步：移动 HEAD
`reset` 做的第一件事是移动 HEAD 的指向。 这与改变 HEAD 自身不同（ `checkout` 所做的）；`reset` 移动 HEAD 指向的分支。 这意味着如果 HEAD 设置为 `master` 分支（例如，你正在 `master` 分支上），运行 `git reset 9e5e64a` 将会使 `master` 指向 `9e5e64a`。

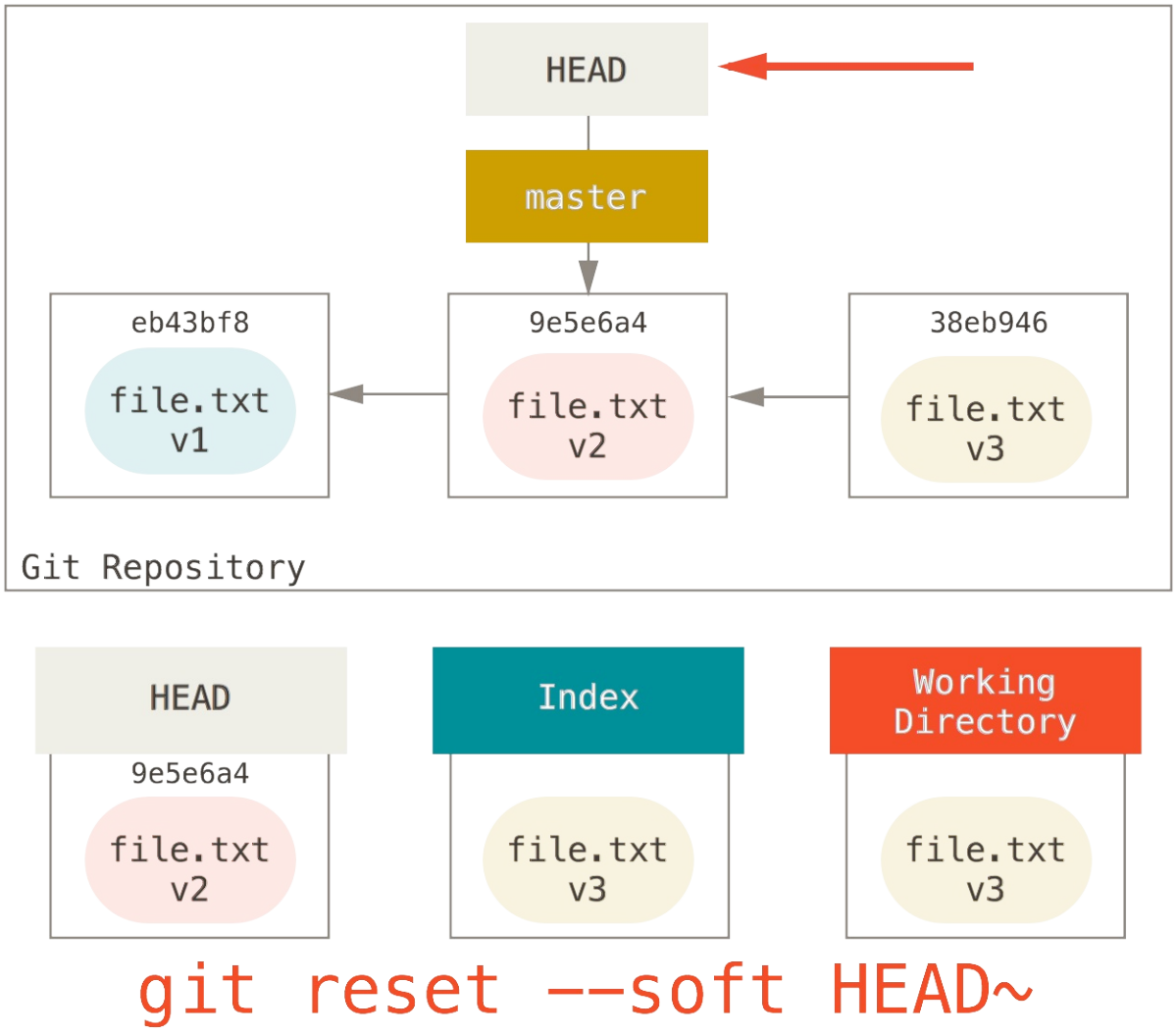


Figure 7-10.

无论你调用了何种形式的带有一个提交的 `reset`，它首先都会尝试这样做。 使用 `reset --soft`，它将仅仅停在那儿。

现在看一眼上图，理解一下发生的事情：它本质上是撤销了上一次 `git commit` 命令。 当你在运行 `git commit` 时，Git 会创建一个新的提交，并移动 HEAD 所指向的分支来使其指向该提交。 当你将它 `reset` 回 `HEAD~`（HEAD 的父结点）时，其实就是把该分支移动回原来的位置，而不会改变索引和工作目录。 现在你可以更新索引并再次运行 `git commit` 来完成 `git commit --amend` 所要做的事情了（见 修改最后一次提交）。

第 2 步：更新索引（--mixed）
注意，如果你现在运行 `git status` 的话，就会看到新的 HEAD 和以绿色标出的它和索引之间的区别。

接下来，`reset` 会用 HEAD 指向的当前快照的内容来更新索引。

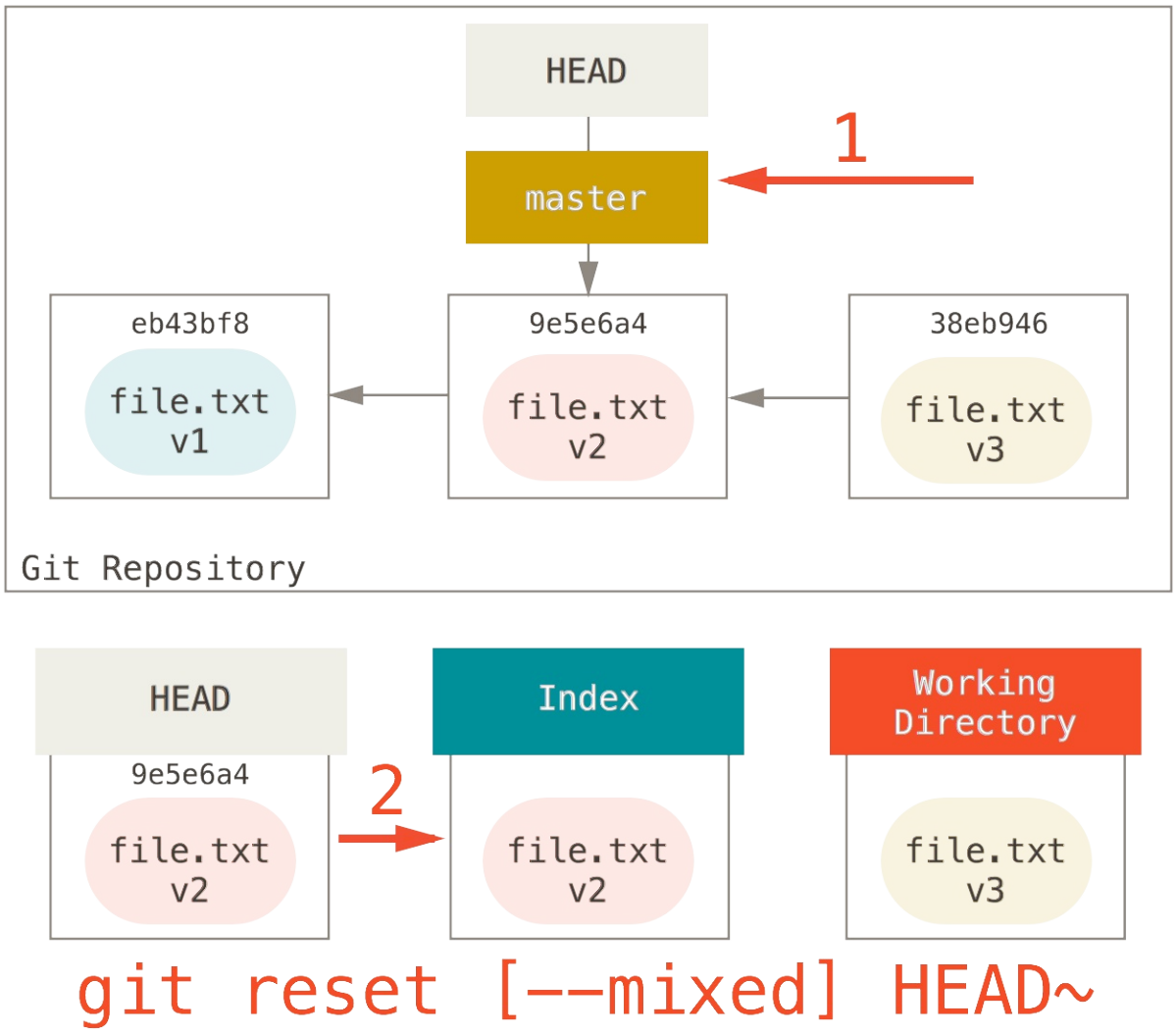


Figure 7-11.

如果指定 `--mixed` 选项，`reset` 将会在这时停止。 这也是默认行为，所以如果没有指定任何选项（在本例中只是 `git reset HEAD~`），这就是命令将会停止的地方。

现在再看一眼上图，理解一下发生的事情：它依然会撤销一上次 `提交`，但还会 取消暂存 所有的东西。于是，我们回滚到了所有 `git add` 和 `git commit` 的命令执行之前。

第 3 步：更新工作目录（--hard）

`reset` 要做的的第三件事情就是让工作目录看起来像索引。 如果使用 `--hard` 选项，它将会继续这一步。

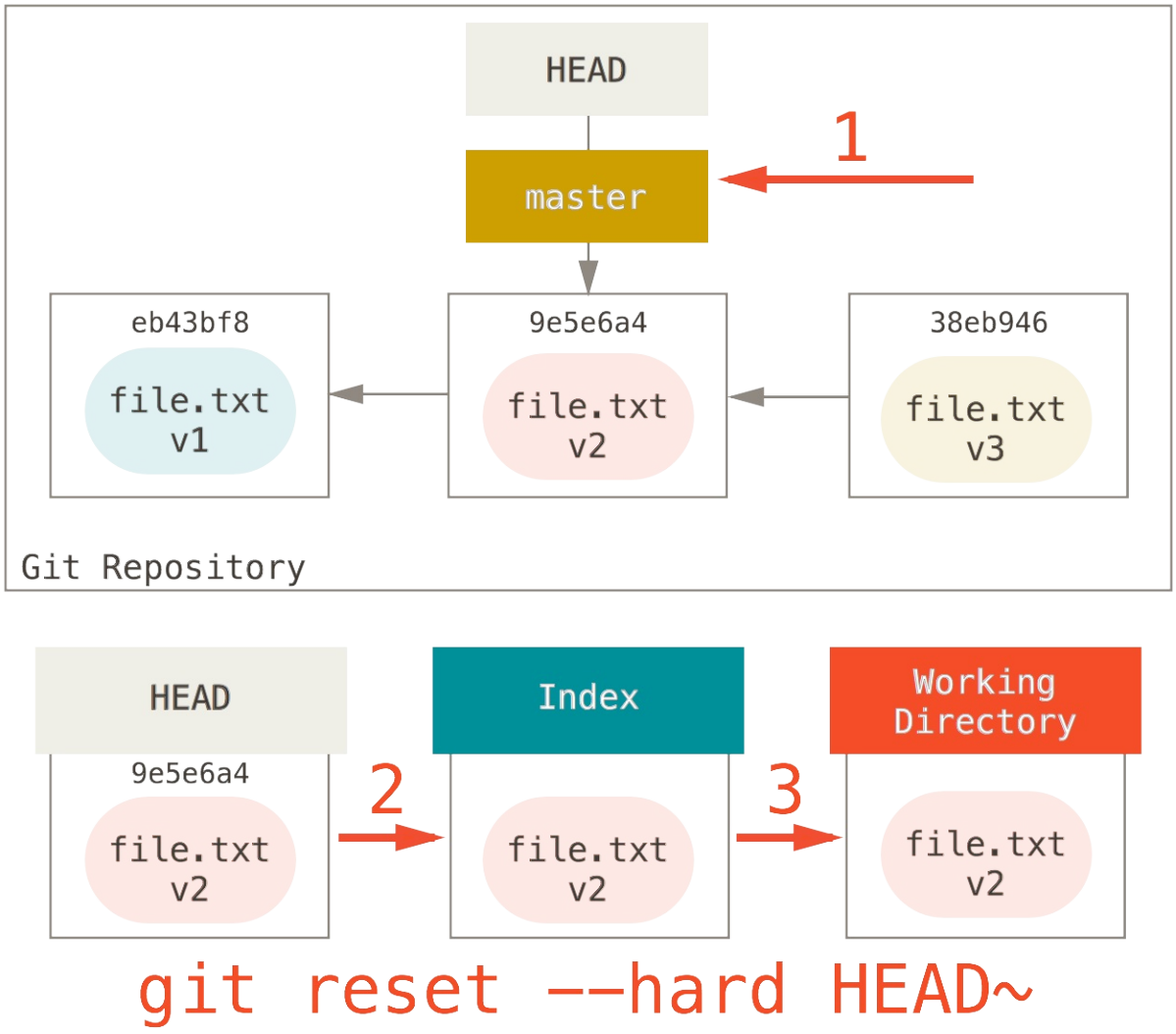


Figure 7-12.

现在让我们回想一下刚才发生的事情。 你撤销了最后的提交、`git add` 和 `git commit` 命令以及工作目录中的所有工作。

必须注意，`--hard` 标记是 `reset` 命令唯一的危险用法，它也是 Git 会真正地销毁数据的仅有的几个操作之一。 其他任何形式的 `reset` 调用都可以轻松撤消，但是 `--hard` 选项不能，因为它强制覆盖了工作目录中的文件。 在这种特殊情况下，我们的 Git 数据库中的一个提交内还留有该文件的 v3 版本，我们可以通过 `reflog` 来找回它。但是若该文件还未提交，Git 仍会覆盖它从而导致无法恢复。

回顾
`reset` 命令会以特定的顺序重写这三棵树，在你指定以下选项时停止：

1. 移动 HEAD 分支的指向（若指定了 `--soft`，则到此停止）

2. 使索引看起来像 HEAD（若未指定 `--hard`，则到此停止）
3. 使工作目录看起来像索引

通过路径来重置

前面讲述了 `reset` 基本形式的行为，不过你还可以给它提供一个作用路径。 若指定了一个路径，`reset` 将会跳过第 1 步，并且将它的作用范围限定为指定的文件或文件集合。 这样做自然有它的道理，因为 HEAD 只是一个指针，你无法让它同时指向两个提交中各自的一部分。 不过索引和工作目录 可以部分更新，所以重置会继续进行第 2、3 步。

现在，假如我们运行 `git reset file.txt`（这其实是 `git reset --mixed HEAD file.txt` 的简写形式，因为你既没有指定一个提交的 SHA-1 或分支，也没有指定 `--soft` 或 `--hard`），它会：

1. 移动 HEAD 分支的指向（已跳过）
2. 让索引看起来像 HEAD（到此处停止）

所以它本质上只是将 `file.txt` 从 HEAD 复制到索引中。

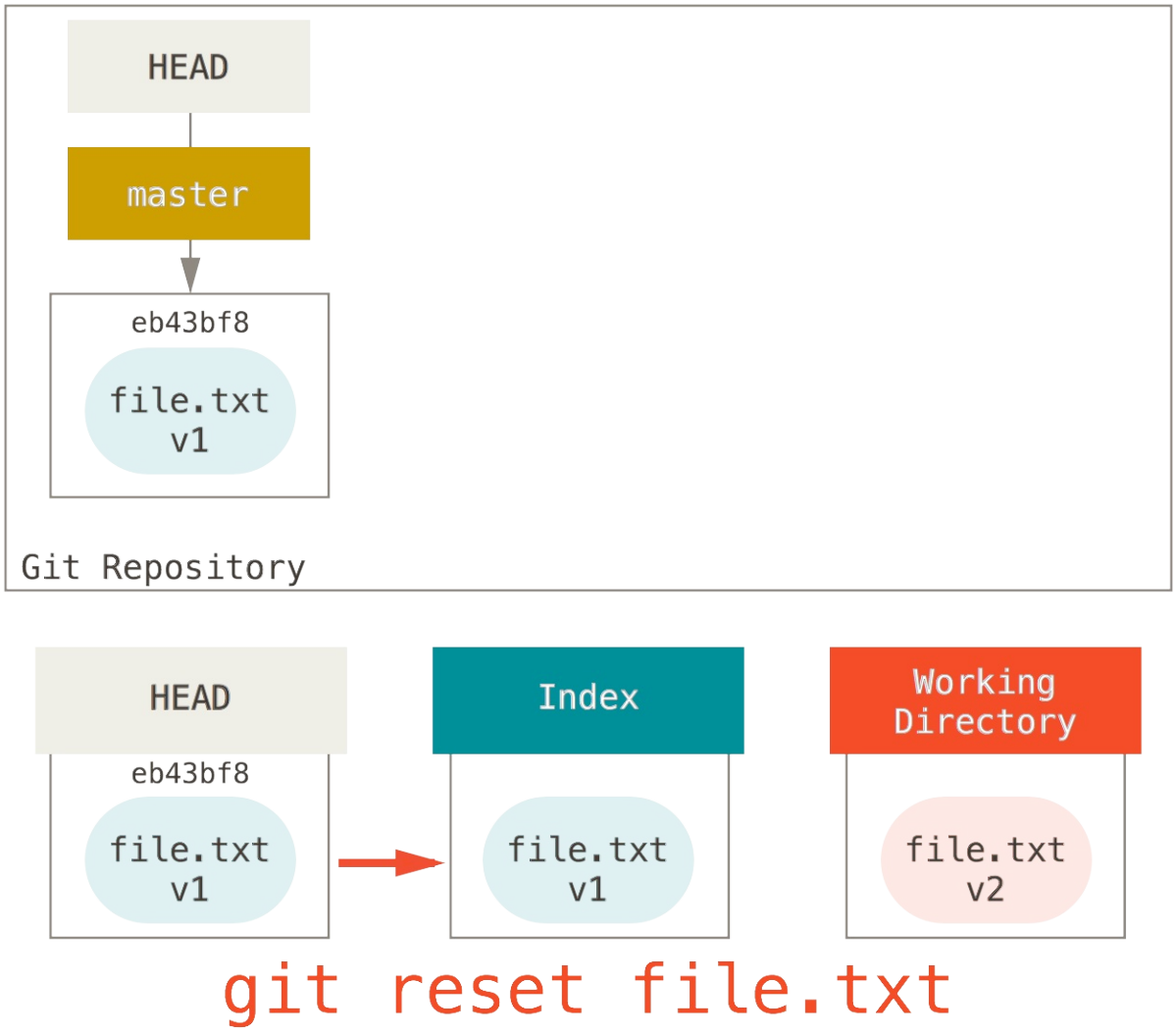


Figure 7-13.

它还有 取消暂存文件 的实际效果。 如果我们查看该命令的示意图，然后再想想 `git add` 所做的事，就会发现它们正好相反。

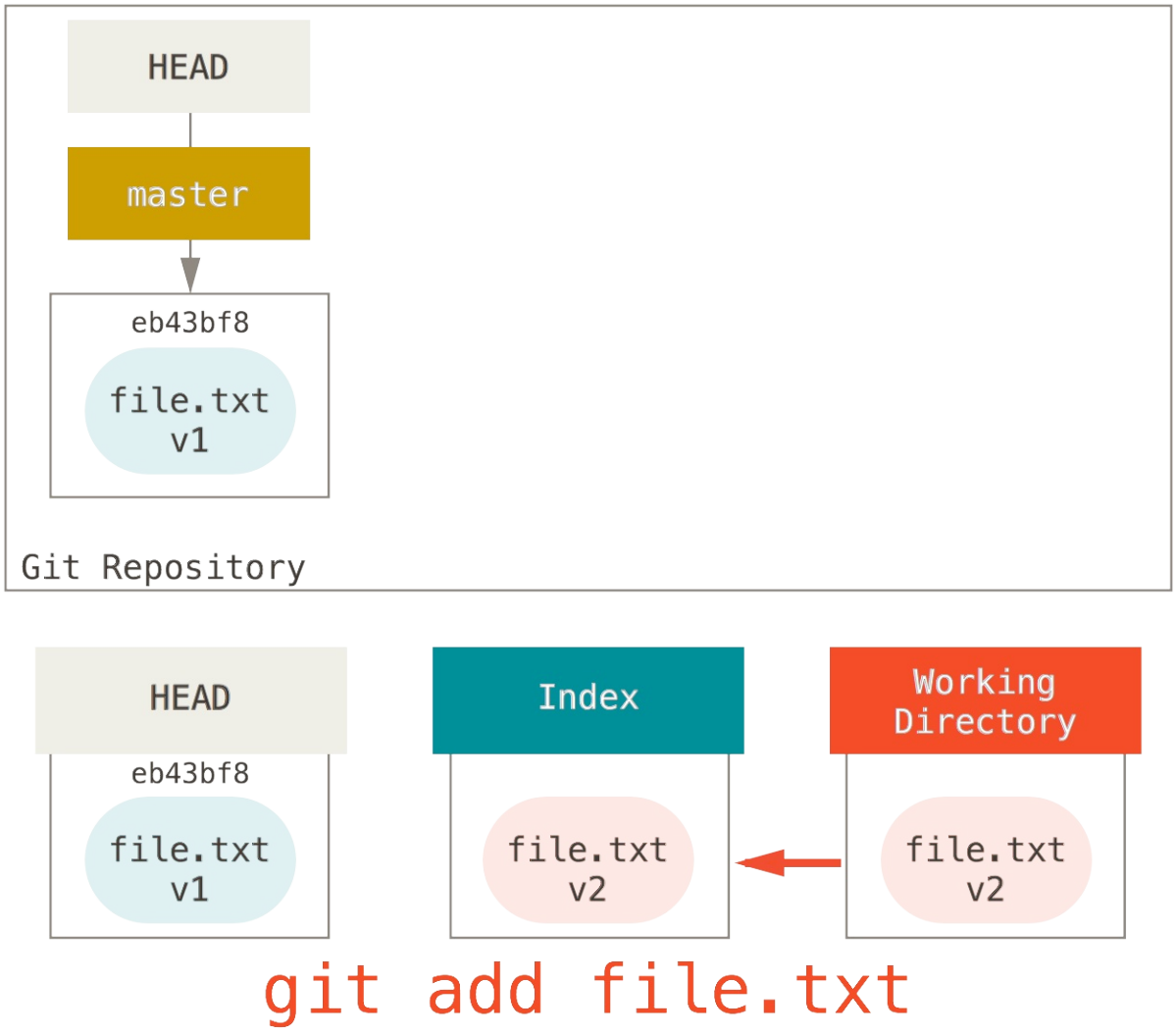


Figure 7-14.

这就是为什么 `git status` 命令的输出会建议运行此命令来取消暂存一个文件。（查看 取消暂存的文件 来了解更多。）

我们可以不让 Git 从 HEAD 拉取数据，而是通过具体指定一个提交来拉取该文件的对应版本。 我们只需运行类似于 `git reset eb43bf file.txt` 的命令即可。

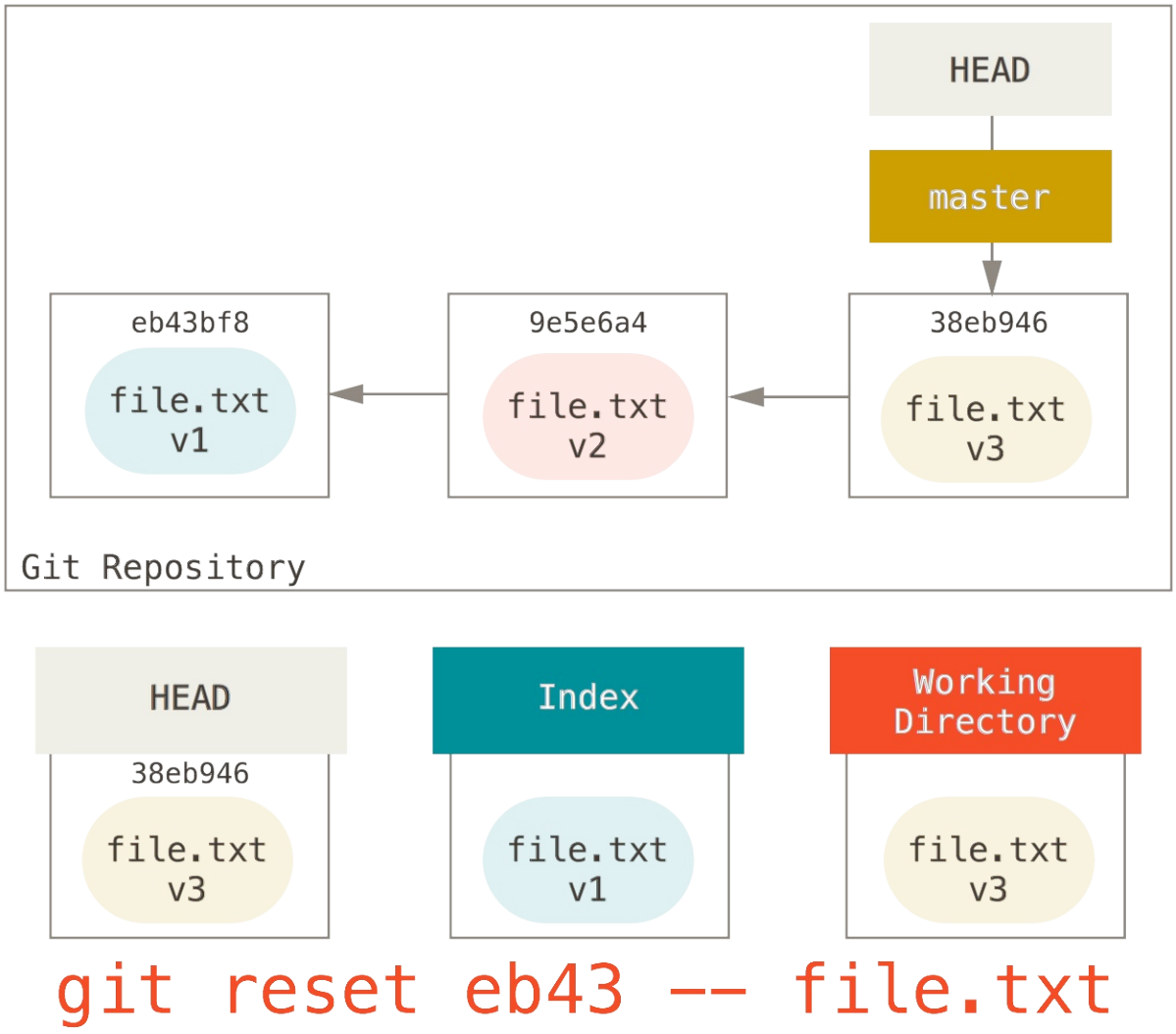


Figure 7-15.

它其实做了同样的事情，也就是把工作目录中的文件恢复到 v1 版本，运行 `git add` 添加它，然后再将它恢复到 v3 版本（只是不用真的过一遍这些步骤）。 如果我们现在运行 `git commit`，它就会记录一条“将该文件恢复到 v1 版本”的更改，尽管我们并未在工作目录中真正地再次拥有它。

还有一点同 `git add` 一样，就是 `reset` 命令也可以接受一个 `--patch` 选项来一块一块地取消暂存的内容。 这样你就可以根据选择来取消暂存或恢复内容了。

压缩

我们来看看如何利用这种新的功能来做一些有趣的事情 - 压缩提交。

假设你的一系列提交信息中有 “oops.”、“WIP” 和 “forgot this file”，聪明的你就能使用 `reset` 来轻松快速地将它们压缩成单个提交，也显出你的聪明。（压缩提交 展示了另一种方式，不过在本例中用 `reset` 更简单。）

假设你有一个项目，第一次提交中有一个文件，第二次提交增加了一个新的文件并修改了第一个文件，第三次提交再次修改了第一个文件。 由于第二次提交是一个未完成的工作，因此你想要压缩它。

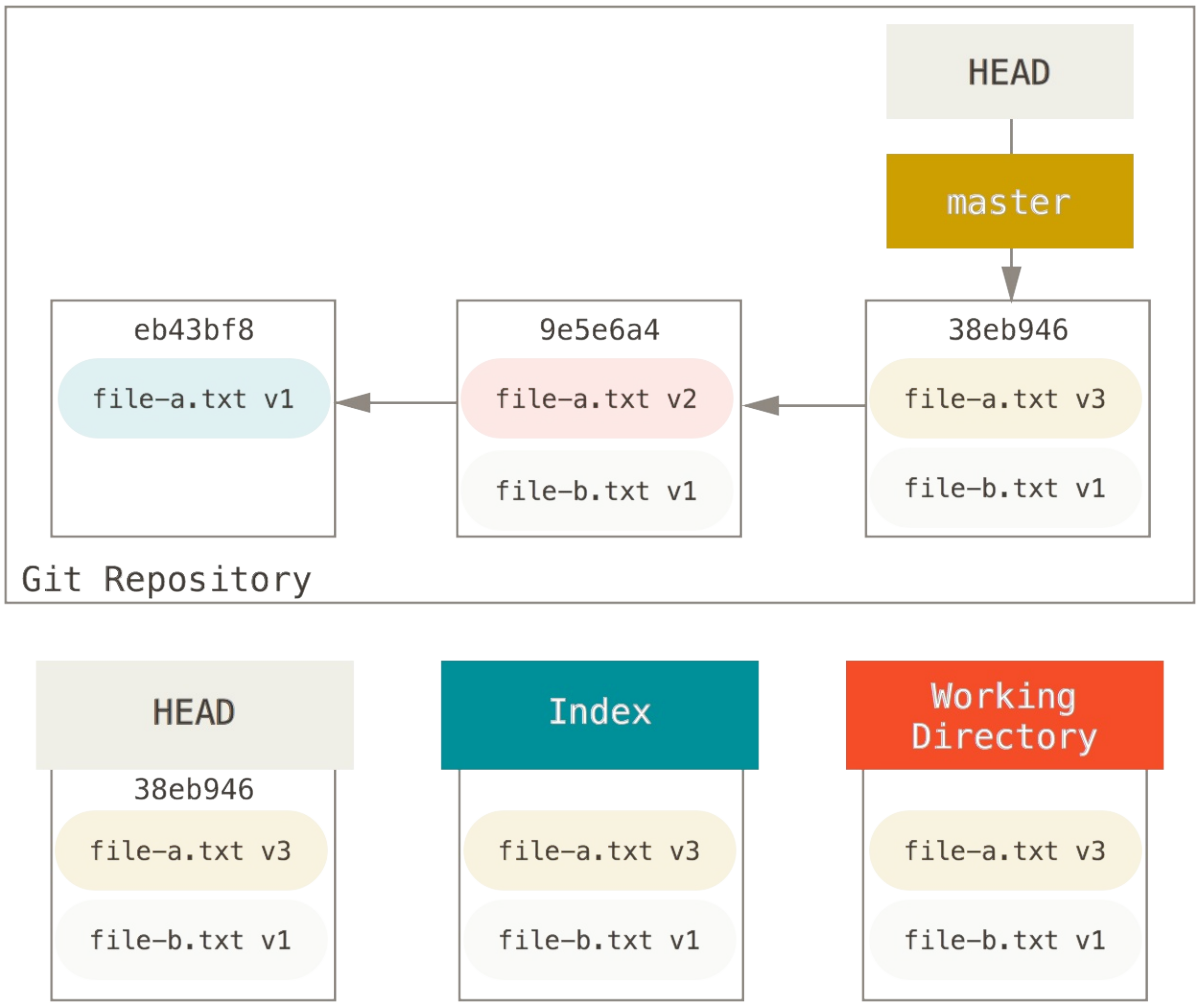


Figure 7-16.

那么可以运行 `git reset --soft HEAD~2` 来将 HEAD 分支移动到一个旧一点的提交上（即你想要保留的第一个提交）：

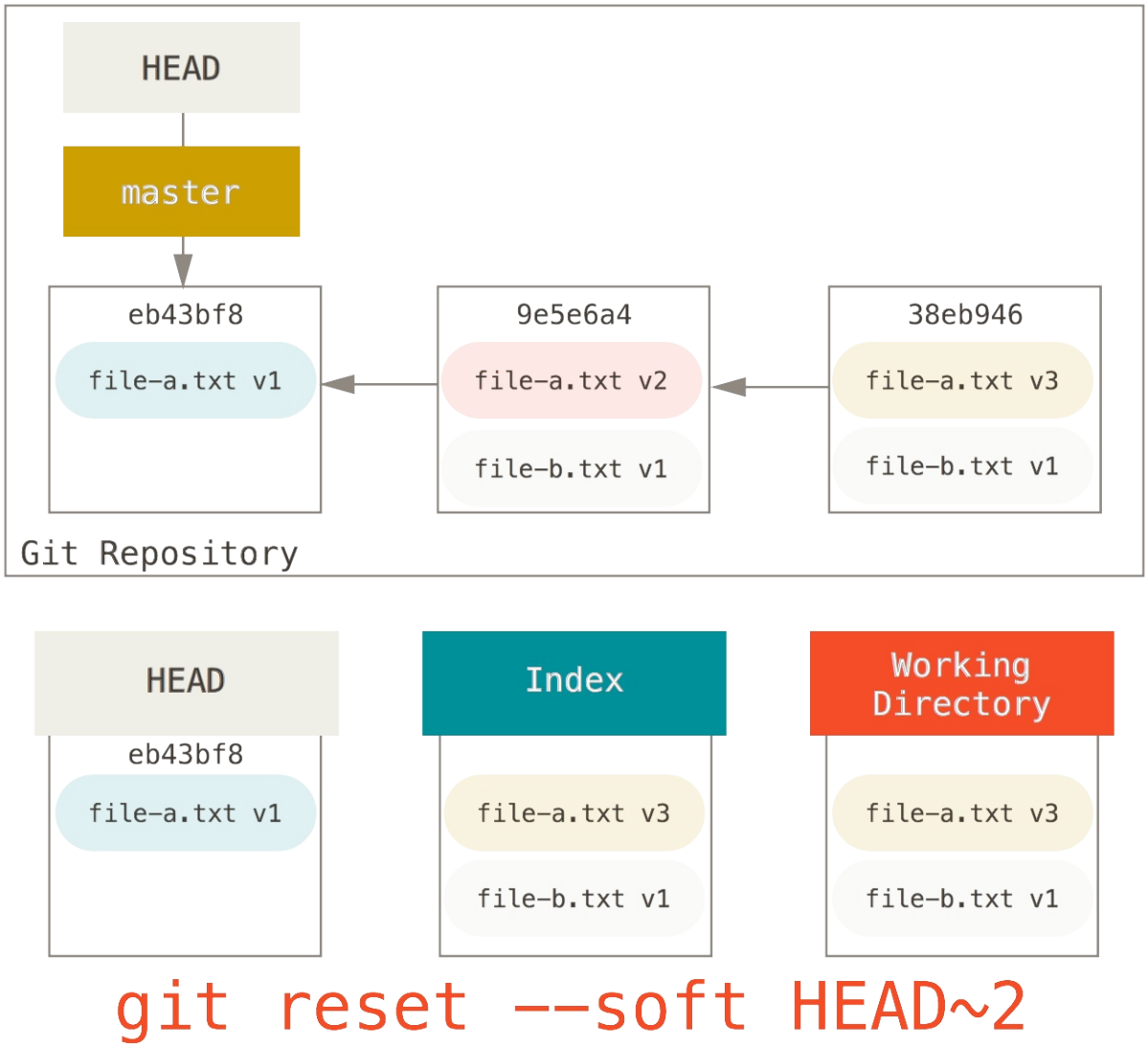


Figure 7-17.

然后只需再次运行 `git commit` ：

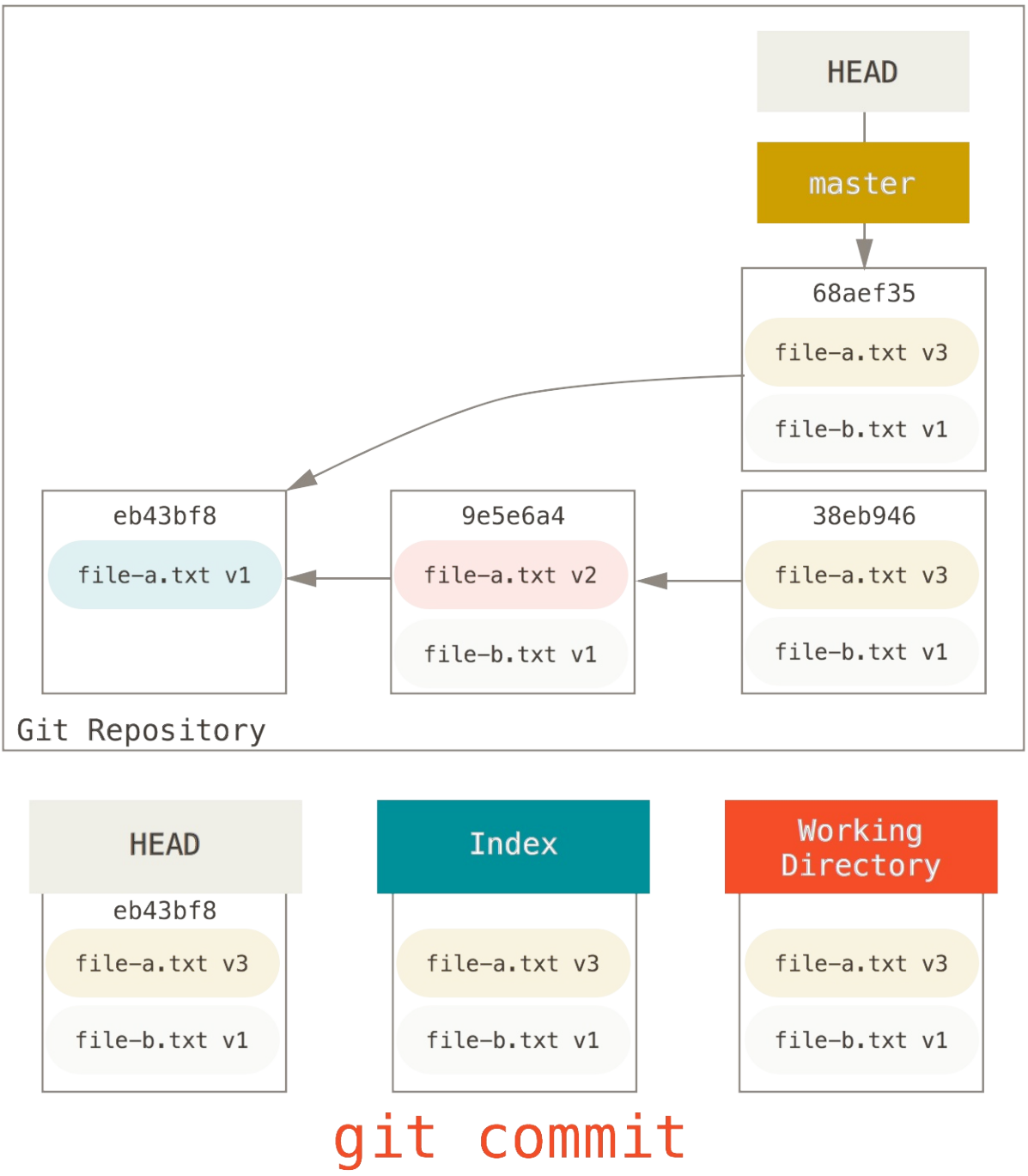


Figure 7-18.

现在你可以查看可到达的历史，即将会推送的历史，现在看起来有个 v1 版 `file-a.txt` 的提交，接着第二个提交将 `file-a.txt` 修改成了 v3 版并增加了 `file-b.txt`。 包含 v2 版本的文件已经不在历史中了。

检出

最后，你大概还想知道 `checkout` 和 `reset` 之间的区别。 和 `reset` 一样， `checkout` 也操纵三棵树，不过它有一点不同，这取决于你是否传给该命令一个文件路径。

不带路径
运行 `git checkout [branch]` 与运行 `git reset --hard [branch]` 非常相似，它会更新所有三棵树使其看起来像 `[branch]` ，不过有两点重要的区别。

首先不同于 `reset --hard` ， `checkout` 对工作目录是安全的，它会通过检查来确保不会将已更改的文件吹走。 其实它还更聪明一些。它会在工作目录中先试着简单合并一下，这样所有还未修改过的文件都会被更新。 而 `reset --hard` 则会不做检查就全面地替换所有东西。

第二个重要的区别是如何更新 HEAD。 `reset` 会移动 HEAD 分支的指向，而 `checkout` 只会移动 HEAD 自身来指向另一个分支。

例如，假设我们有 `master` 和 `develop` 分支，它们分别指向不同的提交；我们现在在 `develop` 上（所以 HEAD 指向它）。 如果我们运行 `git reset master` ，那么 `develop` 自身现在会和 `master` 指向同一个提交。 而如果我们运行 `git checkout master` 的话， `develop` 不会移动， HEAD 自身会移动。 现在 HEAD 将会指向 `master` 。

所以，虽然在这两种情况下我们都移动 HEAD 使其指向了提交 A，但做法是非常不同的。 `reset` 会移动 HEAD 分支的指向，而 `checkout` 则移动 HEAD 自身。

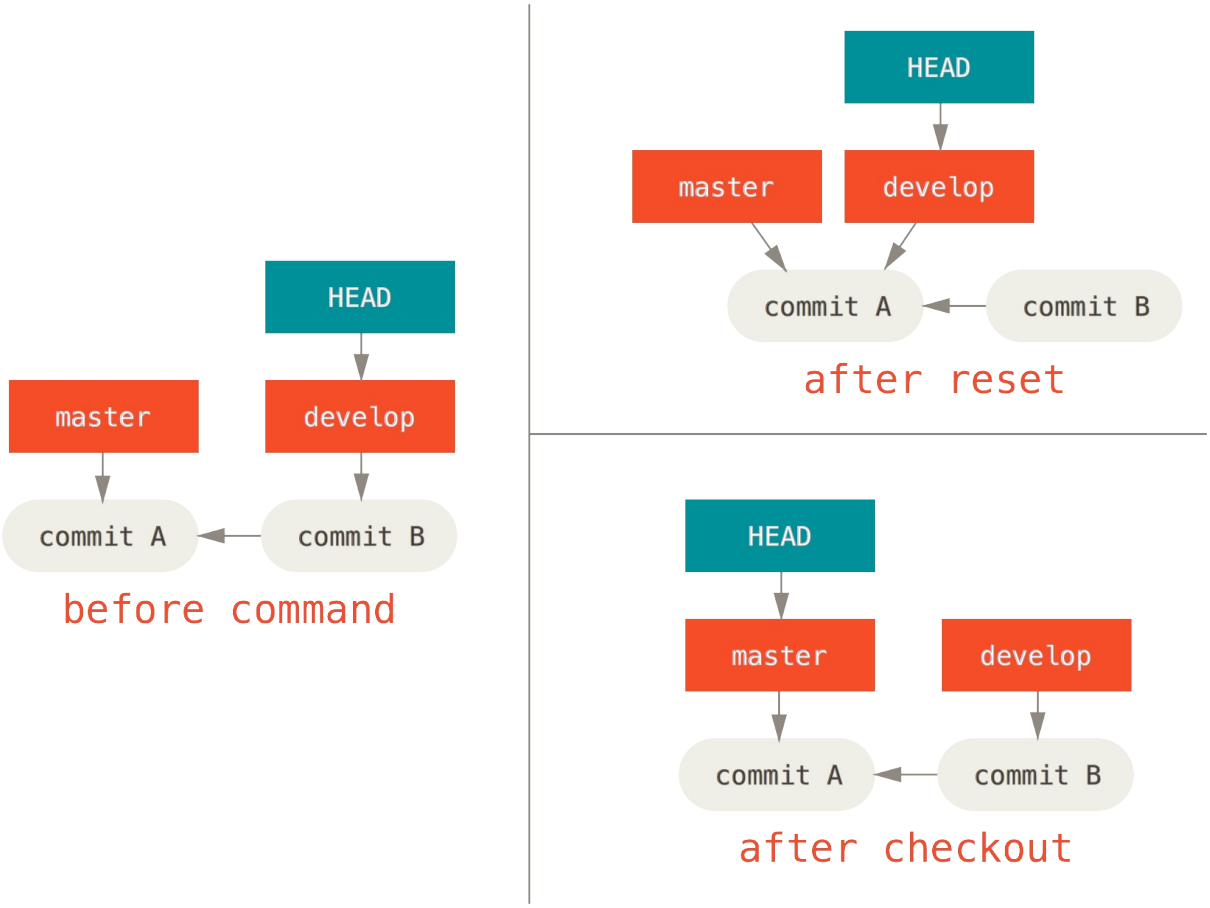


Figure 7-19.

带路径

运行 `checkout` 的另一种方式就是指定一个文件路径，这会像 `reset` 一样不会移动 HEAD。 它就像 `git reset [branch] file` 那样用该次提交中的那个文件来更新索引，但是它也会覆盖工作目录中对应的文件。 它就像是 `git reset --hard [branch] file`（如果 `reset` 允许你这样运行的话）- 这样对工作目录并不安全，它也不会移动 HEAD。

此外，同 `git reset` 和 `git add` 一样，`checkout` 也接受一个 `--patch` 选项，允许你根据选择一块一块地恢复文件内容。

总结

希望你现在熟悉并理解了 `reset` 命令，不过关于它和 `checkout` 之间的区别，你可能还是会有点困惑，毕竟不太可能记住不同调用的所有规则。

下面的速查表列出了命令对树的影响。 “HEAD” 一列中的 “REF” 表示该命令移动了 HEAD 指向的分支引用，而 “HEAD” 则表示只移动了 HEAD 自身。 特别注意 WD Safe? 一列 - 如果它标记为NO，那么运行该命令之前请考虑一下。

HEAD	Index	Workdir	WD Safe?

Commit Level

`reset --soft [commit]`			
REF			
NO			
NO			
YES			
`reset [commit]`			
REF			
YES			
NO			
YES			

|

`reset --hard [commit]`

|

REF

|

YES

|

YES

|

NO

|
|

`checkout [commit]`

|

HEAD

|

YES

|

YES

|

YES

|
|

File Level

| | | | |
|

`reset (commit) [file]`

|

NO

|

YES

|

NO

|

YES

|
|

`checkout (commit) [file]`

|

NO

|

YES

|

YES

|

NO

|

7.8 高级合并

高级合并

在 Git 中合并是相当容易的。 因为 Git 使多次合并另一个分支变得很容易，这意味着你可以有一个始终保持最新的长期分支，经常解决小的冲突，比在一系列提交后解决一个巨大的冲突要好。

然而，有时也会有棘手的冲突。 不像其他的版本控制系统，Git 并不会尝试过于聪明的合并冲突解决方案。 Git 的哲学是聪明地决定无歧义的合并方案，但是如果有冲突，它不会尝试智能地自动解决它。 因此，如果很久之后才合并两个分叉的分支，你可能会撞上一些问题。

在本节中，我们将会仔细查看那些问题是什么以及 Git 给了我们什么工具来帮助我们处理这些更难办的情形。我们也会了解你可以做的不同的、非标准类型的合并，也会看到如何后退到合并之前。

合并冲突

我们在 遇到冲突时的分支合并 介绍了解决合并冲突的一些基础知识，对于更复杂的冲突，Git 提供了几个工具来帮助你指出将会发生什么以及如何更好地处理冲突。

首先，在做一次可能有冲突的合并前尽可能保证工作目录是干净的。 如果你有正在做的工作，要么提交到一个临时分支要么储藏它。 这使你可以撤消在这里尝试做的任何事情。 如果在你尝试一次合并时工作目录中有未保存的改动，下面的这些技巧可能会使你丢失那些工作。

让我们通过一个非常简单的例子来了解一下。 我们有一个超级简单的打印 hello world 的 Ruby 文件。

```
#! /usr/bin/env ruby

def hello
  puts 'hello world'
end

hello()
```

在我们的仓库中，创建一个名为 `whitespace` 的新分支并将所有 Unix 换行符修改为 DOS 换行符，实质上虽然改变了文件的每一行，但改变的都只是空白字符。 然后我们修改行 “hello world” 为 “hello mundo”。

```
$ git checkout -b whitespace
Switched to a new branch 'whitespace'

$ unix2dos hello.rb
unix2dos: converting file hello.rb to DOS format ...
$ git commit -am 'converted hello.rb to DOS'
[whitespace 3270f76] converted hello.rb to DOS
 1 file changed, 7 insertions(+), 7 deletions(-)

$ vim hello.rb
```

```
$ git diff -b
diff --git a/hello.rb b/hello.rb
index ac51efd..e85207e 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,7 @@
 #! /usr/bin/env ruby

 def hello
-  puts 'hello world'
+  puts 'hello mundo'^M
 end

 hello()

$ git commit -am 'hello mundo change'
[whitespace 6d338d2] hello mundo change
 1 file changed, 1 insertion(+), 1 deletion(-)
```

现在我们切换回我们的 `master` 分支并为函数增加一些注释。

```
$ git checkout master
Switched to branch 'master'

$ vim hello.rb
$ git diff
diff --git a/hello.rb b/hello.rb
index ac51efd..36c06c8 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
   puts 'hello world'
 end

$ git commit -am 'document the function'
[master bec6336] document the function
 1 file changed, 1 insertion(+)
```

现在我们尝试合并入我们的 `whitespace` 分支，因为修改了空白字符，所以合并会出现冲突。

```
$ git merge whitespace
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.
```

中断一次合并

我们现在有几个选项。 首先，让我们介绍如何摆脱这个情况。 你可能不想处理冲突这种情况，完全可以通过 `git merge --abort` 来简单地退出合并。

```
$ git status -sb
```

```
## master
UU hello.rb

$ git merge --abort

$ git status -sb
## master
```

`git merge --abort` 选项会尝试恢复到你运行合并前的状态。 但当运行命令前，在工作目录中有未储藏、未提交的修改时它不能完美处理，除此之外它都工作地很好。

如果因为某些原因你发现自己处在一个混乱的状态中然后只是想要重来一次，也可以运行 `git reset --hard HEAD` 回到之前的状态或其他你想要恢复的状态。 请牢记这会将清除工作目录中的所有内容，所以确保你不需要保存这里的任意改动。

忽略空白
在这个特定的例子中，冲突与空白有关。 我们知道这点是因为这个例子很简单，但是在实际的例子中发现这样的冲突也很容易，因为每一行都被移除而在另一边每一行又被加回来了。 默认情况下，Git 认为所有这些行都改动了，所以它不会合并文件。

默认合并策略可以带有参数，其中的几个正好是关于忽略空白改动的。 如果你看到在一次合并中有大量的空白问题，你可以简单地中止它并重做一次，这次使用 `-Xignore-all-space` 或 `-Xignore-space-change` 选项。 第一个选项忽略任意 数量 的已有空白的修改，第二个选项忽略所有空白修改。

```
$ git merge -Xignore-space-change whitespace
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

因为在本例中，实际上文件修改并没有冲突，一旦我们忽略空白修改，每一行都能被很好地合并。

如果你的团队中的某个人可能不小心重新格式化空格为制表符或者相反的操作，这会是一个救命稻草。

手动文件再合并
虽然 Git 对空白的预处理做得很好，还有很多其他类型的修改，Git 也许无法自动处理，但是脚本可以处理它们。 例如，假设 Git 无法处理空白修改因此我们需要手动处理。

我们真正想要做的是对将要合并入的文件在真正合并前运行 `dos2unix` 程序。 所以如果那样的话，我们该如何做？

首先，我们进入到了合并冲突状态。 然后我们想要我的版本的文件，他们的版本的文件（从我们将要合并入的分支）和共同的版本的文件（从分支叉开时的位置）的拷贝。 然后我们想要修复任何一边的文件，并且为这个单独的文件重试一次合并。

获得这三个文件版本实际上相当容易。 Git 在索引中存储了所有这些版本，在 “stages” 下每一个都有一个数字与它们关联。 Stage 1 是它们共同的祖先版本，stage 2 是你的版本，stage 3 来自于 `MERGE_HEAD`，即你将要合并入的版本（ “theirs” ）。

通过 `git show` 命令与一个特别的语法，你可以将冲突文件的这些版本释放出一份拷贝。

```
$ git show :1:hello.rb > hello.common.rb
$ git show :2:hello.rb > hello.ours.rb
$ git show :3:hello.rb > hello.theirs.rb
```

如果你想要更专业一点，也可以使用 `ls-files -u` 底层命令来得到这些文件的 Git blob 对象的实际 SHA-1 值。

```
$ git ls-files -u
100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1       hello.rb
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2       hello.rb
100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3       hello.rb
```

`:1:hello.rb` 只是查找那个 blob 对象 SHA-1 值的简写。

既然在我们的工作目录中已经有这所有三个阶段的内容，我们可以手工修复它们来修复空白问题，然后使用鲜为人知的 `git merge-file` 命令来重新合并那个文件。

```
$ dos2unix hello.theirs.rb
dos2unix: converting file hello.theirs.rb to Unix format ...

$ git merge-file -p \
    hello.ours.rb hello.common.rb hello.theirs.rb > hello.rb

$ git diff -b
diff --cc hello.rb
index 36c06c8,e85207e..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,8 -1,7 +1,8 @@@
  #! /usr/bin/env ruby

 +# prints out a greeting
  def hello
-   puts 'hello world'
+   puts 'hello mundo'
  end

  hello()
```

在这时我们已经漂亮地合并了那个文件。 实际上，这比使用 `ignore-space-change` 选项要更好，因为在合并前真正地修复了空白修改而不是简单地忽略它们。 在使用 `ignore-space-change` 进行合并操作后，我们最终得到了有几行是 DOS 行尾的文件，从而使提交内容混乱了。

如果你想要在最终提交前看一下我们这边与另一边之间实际的修改，你可以使用 `git diff` 来比较将要提交作为合并结果的工作目录与其中任意一个阶段的文件差异。 让我们看看它们。

要在合并前比较结果与在你的分支上的内容，换一句话说，看看合并引入了什么，可以运行 `git diff --ours`

```
$ git diff --ours
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index 36c06c8..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -2,7 +2,7 @@

 # prints out a greeting
 def hello
-  puts 'hello world'
+  puts 'hello mundo'
 end

 hello()
```

这里我们可以很容易地看到在我们的分支上发生了什么，在这次合并中我们实际引入到这个文件的改动，是修改了其中一行。

如果我们想要查看合并的结果与他们那边有什么不同，可以运行 `git diff --theirs`。 在本例及后续的例子中，我们会使用 `-b` 来去除空白，因为我们将它与 Git 中的，而不是我们清理过的 `hello.theirs.rb` 文件比较。

```
$ git diff --theirs -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index e85207e..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
   puts 'hello mundo'
 end
```

最终，你可以通过 `git diff --base` 来查看文件在两边是如何改动的。

```
$ git diff --base -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index ac51efd..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,8 @@
```

```
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
-  puts 'hello world'
+  puts 'hello mundo'
 end

 hello()
```

在这时我们可以使用 `git clean` 命令来清理我们为手动合并而创建但不再有用的额外文件。

```
$ git clean -f
Removing hello.common.rb
Removing hello.ours.rb
Removing hello.theirs.rb
```

检出冲突
也许有时我们并不满意这样的解决方案，或许有时还要手动编辑一边或者两边的冲突，但还是依旧无法正常工作，这时我们需要更多的上下文关联来解决这些冲突。

让我们来稍微改动下例子。 对于本例，我们有两个长期分支，每一个分支都有几个提交，但是在合并时却创建了一个合理的冲突。

```
$ git log --graph --oneline --decorate --all
* f1270f7 (HEAD, master) update README
* 9af9d3b add a README
* 694971d update phrase to hola world
| * e3eb223 (mundo) add more tests
| * 7cff591 add testing script
| * c3ffff1 changed text to hello mundo
|/
* b7dcc89 initial hello world code
```

现在有只在 `master` 分支上的三次单独提交，还有其他三次提交在 `mundo` 分支上。 如果我们尝试将 `mundo` 分支合并入 `master` 分支，我们得到一个冲突。

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.
```

我们想要看一下合并冲突是什么。 如果我们打开这个文件，我们将会看到类似下面的内容：

```
#! /usr/bin/env ruby

def hello
<<<<<<< HEAD
  puts 'hola world'
```

```
=======
  puts 'hello mundo'
>>>>>>> mundo
end

hello()
```

合并的两边都向这个文件增加了内容，但是导致冲突的原因是其中一些提交修改了文件的同一个地方。

让我们探索一下现在你手边可用来查明这个冲突是如何产生的工具。 应该如何修复这个冲突看起来或许并不明显。 这时你需要更多上下文。

一个很有用的工具是带 `--conflict` 选项的 `git checkout`。 这会重新检出文件并替换合并冲突标记。 如果想要重置标记并尝试再次解决它们的话这会很有用。

可以传递给 `--conflict` 参数 `diff3` 或 `merge`（默认选项）。 如果传给它 `diff3`，Git 会使用一个略微不同版本的冲突标记：不仅仅只给你 “ours” 和 “theirs” 版本，同时也会有 “base” 版本在中间来给你更多的上下文。

```
$ git checkout --conflict=diff3 hello.rb
```

一旦我们运行它，文件看起来会像下面这样：

```
#! /usr/bin/env ruby

def hello
<<<<<<< ours
  puts 'hola world'
||||||| base
  puts 'hello world'
=======
  puts 'hello mundo'
>>>>>>> theirs
end

hello()
```

如果你喜欢这种格式，可以通过设置 `merge.conflictstyle` 选项为 `diff3` 来做为以后合并冲突的默认选项。

```
$ git config --global merge.conflictstyle diff3
```

`git checkout` 命令也可以使用 `--ours` 和 `--theirs` 选项，这是一种无需合并的快速方式，你可以选择留下一边的修改而丢弃掉另一边修改。

当有二进制文件冲突时这可能会特别有用，因为可以简单地选择一边，或者可以只合并另一个分支的特定

文件 - 可以做一次合并然后在提交前检出一边或另一边的特定文件。

合并日志
另一个解决合并冲突有用的工具是 `git log`。 这可以帮助你得到那些对冲突有影响的上下文。 回顾一点历史来记起为什么两条线上的开发会触碰同一片代码有时会很有用。

为了得到此次合并中包含的每一个分支的所有独立提交的列表，我们可以使用之前在 三点 学习的 “三点” 语法。

```
$ git log --oneline --left-right HEAD...MERGE_HEAD
< f1270f7 update README
< 9af9d3b add a README
< 694971d update phrase to hola world
> e3eb223 add more tests
> 7cff591 add testing script
> c3ffff1 changed text to hello mundo
```

这个漂亮的列表包含 6 个提交和每一个提交所在的不同开发路径。

我们可以通过更加特定的上下文来进一步简化这个列表。 如果我们添加 `--merge` 选项到 `git log` 中，它会只显示任何一边接触了合并冲突文件的提交。

```
$ git log --oneline --left-right --merge
< 694971d update phrase to hola world
> c3ffff1 changed text to hello mundo
```

如果你运行命令时用 `-p` 选项代替，你会得到所有冲突文件的区别。 快速获得你需要帮助理解为什么发生冲突的上下文，以及如何聪明地解决它，这会 非常 有用。

组合式差异格式
因为 Git 暂存合并成功的结果，当你在合并冲突状态下运行 `git diff` 时，只会得到现在还在冲突状态的区别。 当需要查看你还需要解决哪些冲突时这很有用。

在合并冲突后直接运行的 `git diff` 会给你一个相当独特的输出格式。

```
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,11 @@@
  #! /usr/bin/env ruby

  def hello
++<<<<<<< HEAD
 +  puts 'hola world'
++=======
+   puts 'hello mundo'
```

```
++>>>>>>> mundo
  end

  hello()
```

这种叫作 “组合式差异” 的格式会在每一行给你两列数据。 第一列为你显示 “ours” 分支与工作目录的文件区别（添加或删除），第二列显示 “theirs” 分支与工作目录的拷贝区别。

所以在上面的例子中可以看到 `<<<<<<<` 与 `>>>>>>>` 行在工作拷贝中但是并不在合并的任意一边中。这很有意义，合并工具因为我们的上下文被困住了，它期望我们去移除它们。

如果我们解决冲突再次运行 `git diff` ，我们将会看到同样的事情，但是它有一点帮助。

```
$ vim hello.rb
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
 -  puts 'hello mundo'
++  puts 'hola mundo'
  end

  hello()
```

这里显示出 “hola world” 在我们这边但不在工作拷贝中，那个 “hello mundo” 在他们那边但不在工作拷贝中，最终 “hola mundo” 不在任何一边但是现在在工作拷贝中。 在提交解决方案前这对审核很有用。

也可以在合并后通过 `git log` 来获取相同信息，并查看冲突是如何解决的。 如果你对一个合并提交运行 `git show` 命令 Git 将会输出这种格式，或者你也可以在 `git log -p`（默认情况下该命令只会展示还没有合并的补丁）命令之后加上 `--cc` 选项。

```
$ git log --cc -p -1
commit 14f41939956d80b9e17bb8721354c33f8d5b5a79
Merge: f1270f7 e3eb223
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Sep 19 18:14:49 2014 +0200

    Merge branch 'mundo'

    Conflicts:
        hello.rb

diff --cc hello.rb
index 0399cd5,59727f0..e1d0799
--- a/hello.rb
```

```
+++ b/hello.rb
@@ -1,7 -1,7 +1,7 @@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
-  puts 'hello mundo'
++  puts 'hola mundo'
  end

  hello()
```

撤消合并

虽然你已经知道如何创建一个合并提交，但有时出错是在所难免的。 使用 Git 最棒的一件事情是犯错是可以的，因为有可能（大多数情况下都很容易）修复它们。

合并提交并无不同。 假设现在在一个特性分支上工作，不小心将其合并到 `master` 中，现在提交历史看起来是这样：

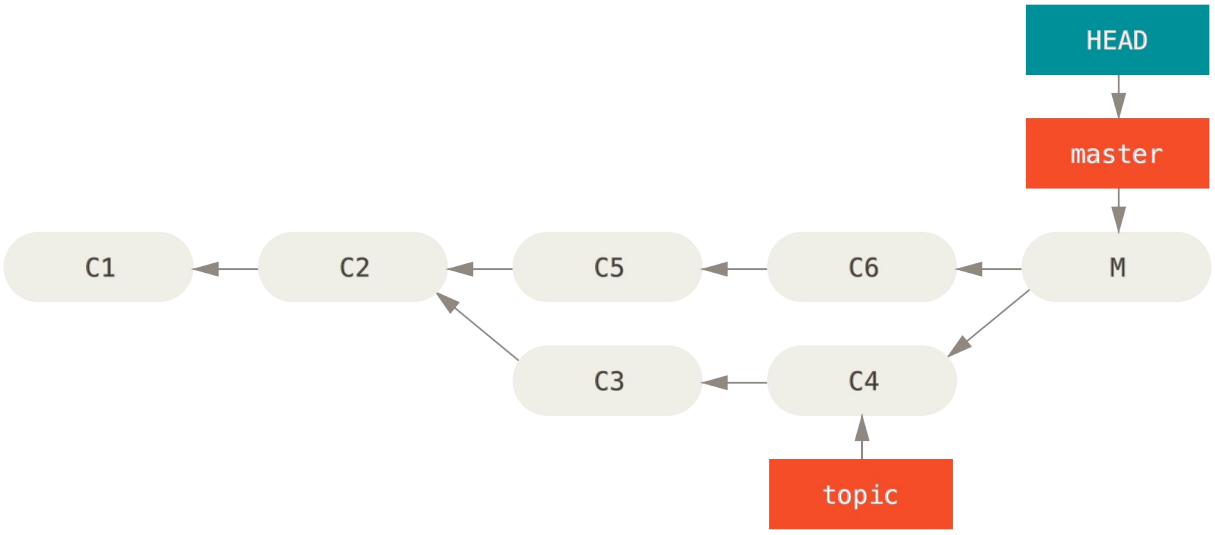


Figure 7-20. 意外的合并提交

有两种方法来解决这个问题，这取决于你想要的结果是什么。

修复引用

如果这个不想要的合并提交只存在于你的本地仓库中，最简单且最好的解决方案是移动分支到你想要它指向的地方。 大多数情况下，如果你在错误的 `git merge` 后运行 `git reset --hard HEAD~`，这会重置分支指向所以它们看起来像这样：

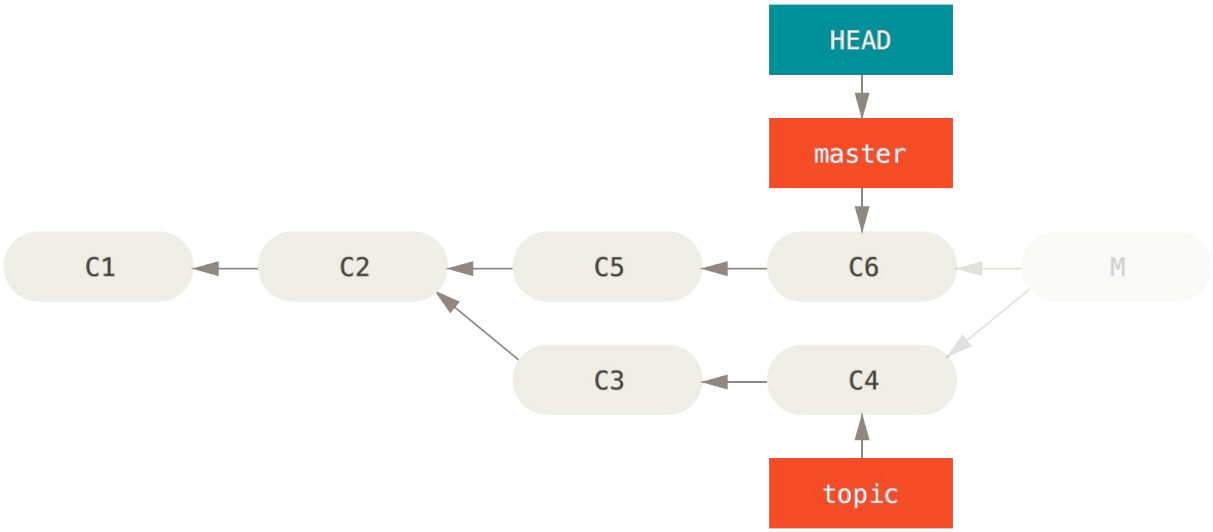


Figure 7-21. 在 `git reset --hard HEAD~` 之后的历史

我们之前在 重置揭密 已经介绍了 `reset`，所以现在指出这里发生了什么并不是很困难。 让我们快速复习下：`reset --hard` 通常会经历三步：

1. 移动 HEAD 指向的分支。 在本例中，我们想要移动 `master` 到合并提交（ `C6` ）之前所在的位置。
2. 使索引看起来像 HEAD。
3. 使工作目录看起来像索引。

这个方法的缺点是它会重写历史，在一个共享的仓库中这会造成问题的。 查阅 变基的风险 来了解更多可能发生的事情；用简单的话说就是如果其他人已经有你将要重写的提交，你应当避免使用 `reset`。 如果有任何其他提交在合并之后创建了，那么这个方法也会无效；移动引用实际上会丢失那些改动。

还原提交
如果移动分支指针并不适合你，Git 给你一个生成一个新提交的选项，提交将会撤消一个已存在提交的所有修改。 Git 称这个操作为 “还原” ，在这个特定的场景下，你可以像这样调用它：

```
$ git revert -m 1 HEAD
[master b1d8379] Revert "Merge branch 'topic'"
```

`-m 1` 标记指出 “mainline” 需要被保留下来的父结点。 当你引入一个合并到 `HEAD`（ `git merge topic` ），新提交有两个父结点：第一个是 `HEAD`（ `C6` ），第二个是将要合并入分支的最新提交（ `C4` ）。 在本例中，我们想要撤消所有由父结点 #2（ `C4` ）合并引入的修改，同时保留从父结点 #1（ `C4` ）开始的所有内容。

有还原提交的历史看起来像这样：

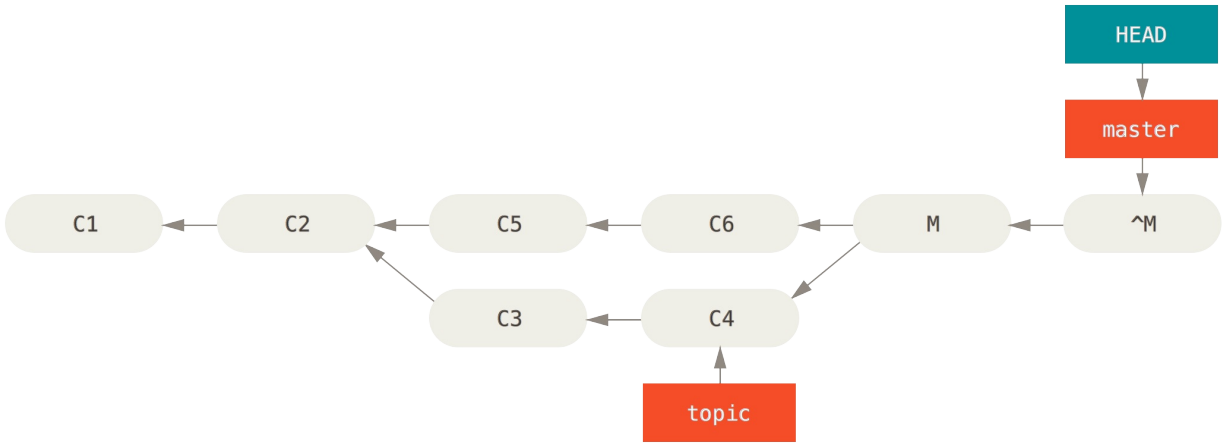


Figure 7-22. 在 `git revert -m 1` 后的历史

新的提交 `^M` 与 `C6` 有完全一样的内容，所以从这儿开始就像合并从未发生过，除了“现在还没合并”的提交依然在 `HEAD` 的历史中。 如果你尝试再次合并 `topic` 到 `master` Git 会感到困惑：

```
$ git merge topic
Already up-to-date.
```

`topic` 中并没有东西不能从 `master` 中追踪到达。 更糟的是，如果你在 `topic` 中增加工作然后再次合并，Git 只会引入被还原的合并 之后 的修改。

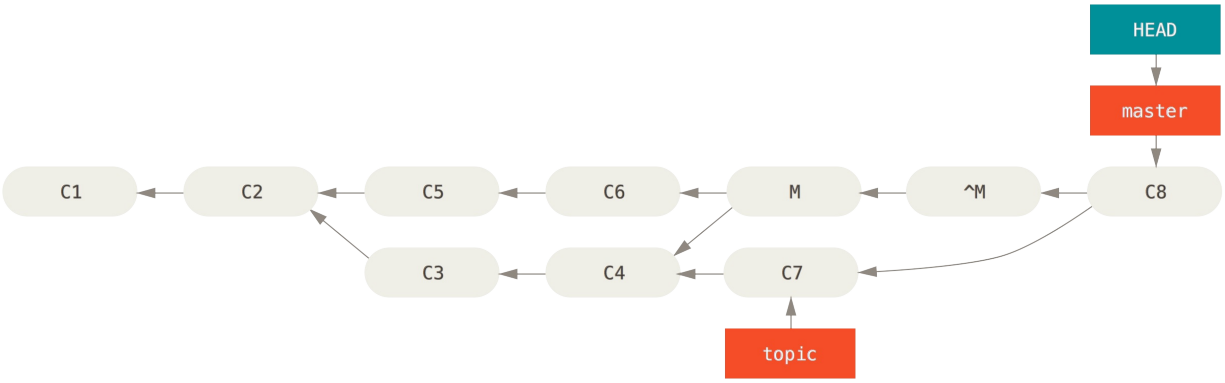


Figure 7-23. 含有坏掉合并的历史

解决这个最好的方式是撤消还原原始的合并，因为现在你想要引入被还原出去的修改，然后 创建一个新的合并提交：

```
$ git revert ^M
[master 09f0126] Revert "Revert "Merge branch 'topic'""
```

```
$ git merge topic
```

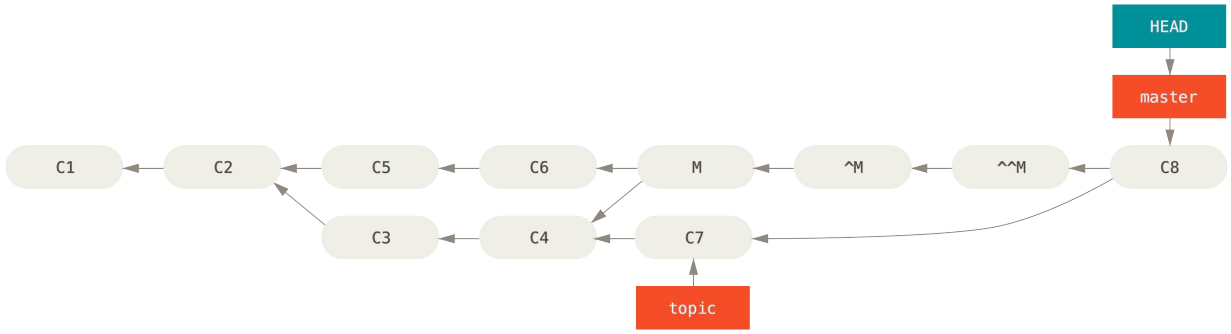


Figure 7-24. 在重新合并一个还原合并后的历史

在本例中，`M` 与 `^M` 抵消了。 `^^M` 事实上合并入了 `C3` 与 `C4` 的修改，`C8` 合并了 `C7` 的修改，所以现在 `topic` 已经完全被合并了。

其他类型的合并

到目前为止我们介绍的都是通过一个叫作 “recursive” 的合并策略来正常处理的两个分支的正常合并。然而还有其他方式来合并两个分支到一起。 让我们来快速介绍其中的几个。

我们的或他们的偏好

首先，有另一种我们可以通过 “recursive” 合并模式做的有用工作。 我们之前已经看到传递给 `-X` 的 `ignore-all-space` 与 `ignore-space-change` 选项，但是我们也可以告诉 Git 当它看见一个冲突时直接选择一边。

默认情况下，当 Git 看到两个分支合并中的冲突时，它会将合并冲突标记添加到你的代码中并标记文件为冲突状态来让你解决。 如果你希望 Git 简单地选择特定的一边并忽略另外一边而不是让你手动合并冲突，你可以传递给 `merge` 命令一个 `-Xours` 或 `-Xtheirs` 参数。

如果 Git 看到这个，它并不会增加冲突标记。 任何可以合并的区别，它会直接合并。 任何有冲突的区别，它会简单地选择你全局指定的一边，包括二进制文件。

如果我们回到之前我们使用的 “hello world” 例子中，我们可以看到合并入我们的分支时引发了冲突。

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Automatic merge failed; fix conflicts and then commit the result.
```

然而如果我们运行时增加 `-Xours` 或 `-Xtheirs` 参数就不会有冲突。

```
$ git merge -Xours mundo
```

```
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 +-
 test.sh  | 2 ++
 2 files changed, 3 insertions(+), 1 deletion(-)
 create mode 100644 test.sh
```

在上例中，它并不会为 “hello mundo” 与 “hola world” 标记合并冲突，它只会简单地选取 “hola world” 。 然而，在那个分支上所有其他非冲突的改动都可以被成功地合并入。

这个选项也可以传递给我们之前看到的 `git merge-file` 命令，通过运行类似 `git merge-file --ours` 的命令来合并单个文件。

如果想要做类似的事情但是甚至并不想让 Git 尝试合并另外一边的修改，有一个更严格的选项，它是 “ours” 合并 策略。 这与 “ours” recursive 合并 选项 不同。

这本质上会做一次假的合并。 它会记录一个以两边分支作为父结点的新合并提交，但是它甚至根本不关注你正合并入的分支。 它只会简单地把当前分支的代码当作合并结果记录下来。

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
$
```

你可以看到合并后与合并前我们的分支并没有任何区别。

当再次合并时从本质上欺骗 Git 认为那个分支已经合并过经常是很有用的。 例如，假设你有一个分叉的 `release` 分支并且在上面做了一些你想要在未来某个时候合并回 `master` 的工作。 与此同时 `master` 分支上的某些 bugfix 需要向后移植回 `release` 分支。 你可以合并 bugfix 分支进入 `release` 分支同时也 `merge -s ours` 合并进入你的 `master` 分支（即使那个修复已经在那儿了）这样当你之后再次合并 `release` 分支时，就不会有来自 bugfix 的冲突。

子树合并

子树合并的思想是你有两个项目，并且其中一个映射到另一个项目的一个子目录，或者反过来也行。 当你执行一个子树合并时，Git 通常可以自动计算出其中一个是另外一个的子树从而实现正确的合并。

我们来看一个例子如何将一个项目加入到一个已存在的项目中，然后将第二个项目的代码合并到第一个项目的子目录中。

首先，我们将 Rack 应用添加到你的项目里。 我们把 Rack 项目作为一个远程的引用添加到我们的项目里，然后检出到它自己的分支。

```
$ git remote add rack_remote https://github.com/rack/rack
$ git fetch rack_remote
warning: no common commits
remote: Counting objects: 3184, done.
```

```
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
From https://github.com/rack/rack
 * [new branch]      build      -> rack_remote/build
 * [new branch]      master     -> rack_remote/master
 * [new branch]      rack-0.4   -> rack_remote/rack-0.4
 * [new branch]      rack-0.9   -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch refs/remotes/rack_remote/master.
Switched to a new branch "rack_branch"
```

现在在我们的 `rack_branch` 分支里就有 Rack 项目的根目录，而我们的项目则在 `master` 分支里。 如果你从一个分支切换到另一个分支，你可以看到它们的项目根目录是不同的：

```
$ ls
AUTHORS         KNOWN-ISSUES   Rakefile      contrib         lib
COPYING         README         bin           example         test
$ git checkout master
Switched to branch "master"
$ ls
README
```

这个是一个比较奇怪的概念。 并不是仓库中的所有分支都是必须属于同一个项目的分支. 这并不常见，因为没啥用，但是却是在不同分支里包含两条完全不同提交历史的最简单的方法。

在这个例子中，我们希望将 Rack 项目拉到 `master` 项目中作为一个子目录。 我们可以在 Git 中执行 `git read-tree` 来实现。 你可以在 Git 内部原理 中查看更多 `read-tree` 的相关信息，现在你只需要知道它会读取一个分支的根目录树到当前的暂存区和工作目录里。 先切回你的 `master` 分支，将 `rack_back` 分支拉取到我们项目的 `master` 分支中的 `rack` 子目录。

```
$ git read-tree --prefix=rack/ -u rack_branch
```

当我们提交时，那个子目录中拥有所有 Rack 项目的文件 —— 就像我们直接从压缩包里复制出来的一样。 有趣的是你可以很容易地将一个分支的变更合并到另一个分支里。 所以，当 Rack 项目有更新时，我们可以切换到那个分支来拉取上游的变更。

```
$ git checkout rack_branch
$ git pull
```

接着，我们可以将这些变更合并回我们的 `master` 分支。 使用 `--squash` 选项和使用 `-Xsubtree` 选项（它采用递归合并策略），都可以用来可以拉取变更并且预填充提交信息。 （递归策略在这里是默认的，提到它是为了让读者有个清晰的概念。）

```
$ git checkout master
```

```
$ git merge --squash -s recursive -Xsubtree=rack rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

Rack 项目中所有的改动都被合并了，等待被提交到本地。 你也可以用相反的方法——在 master 分支上的 `rack` 子目录中做改动然后将它们合并入你的 `rack_branch` 分支中，之后你可能将其提交给项目维护着或者将它们推送到上游。

这给我们提供了一种类似子模块工作流的工作方式，但是它并不需要用到子模块（有关子模块的内容我们会在 子模块 中介绍）。 我们可以在自己的仓库中保持一些和其他项目相关的分支，偶尔使用子树合并将它们合并到我们的项目中。 某些时候这种方式很有用，例如当所有的代码都提交到一个地方的时候。 然而，它同时也有缺点，它更加复杂且更容易让人犯错，例如重复合并改动或者不小心将分支提交到一个无关的仓库上去。

另外一个有点奇怪的地方是，当你想查看 `rack` 子目录和 `rack_branch` 分支的差异——来确定你是否需要合并它们——你不能使用普通的 `diff` 命令。 取而代之的是，你必须使用 `git diff-tree` 来和你的目标分支做比较：

```
$ git diff-tree -p rack_branch
```

或者，将你的 `rack` 子目和最近一次从服务器上抓取的 `master` 分支进行比较，你可以运行：

```
$ git diff-tree -p rack_remote/master
```

7.9 Rerere

Rerere

`git rerere` 功能是一个隐藏的功能。 正如它的名字 “reuse recorded resolution” 所指，它允许你让 Git 记住解决一个块冲突的方法，这样在下一次看到相同冲突时，Git 可以为你自动地解决它。

有几种情形下这个功能会非常有用。 在文档中提到的一个例子是如果你想要保证一个长期分支会干净地合并，但是又不想要一串中间的合并提交。 将 `rerere` 功能打开后偶尔合并，解决冲突，然后返回到合并前。 如果你持续这样做，那么最终的合并会很容易，因为 `rerere` 可以为你自动做所有的事情。

可以将同样的策略用在维持一个变基的分支时，这样就不用每次解决同样的变基冲突了。 或者你将一个分支合并并修复了一堆冲突后想要用变基来替代合并 - 你可能并不想要再次解决相同的冲突。

另一个情形是当你偶尔将一堆正在改进的特性分支合并到一个可测试的头时，就像 Git 项目自身经常做的。 如果测试失败，你可以倒回合并之前然后在去除导致测试失败的那个特性分支后重做合并，而不用再次重新解决所有的冲突。

为了启用 `rerere` 功能，仅仅需要运行这个配置选项：

```
$ git diff-tree -p rack_remote/master
```

也通过在特定的仓库中创建 `.git/rr-cache` 目录来开启它，但是设置选项更干净并且可以应用到全局。

现在我们看一个简单的例子，类似之前的那个。 假设有一个像这样的文件：

```
#! /usr/bin/env ruby

def hello
  puts 'hello world'
end
```

在一个分支中修改单词 “hello” 为 “hola” ，然后在另一个分支中修改 “world” 为 “mundo” ，就像之前一样。

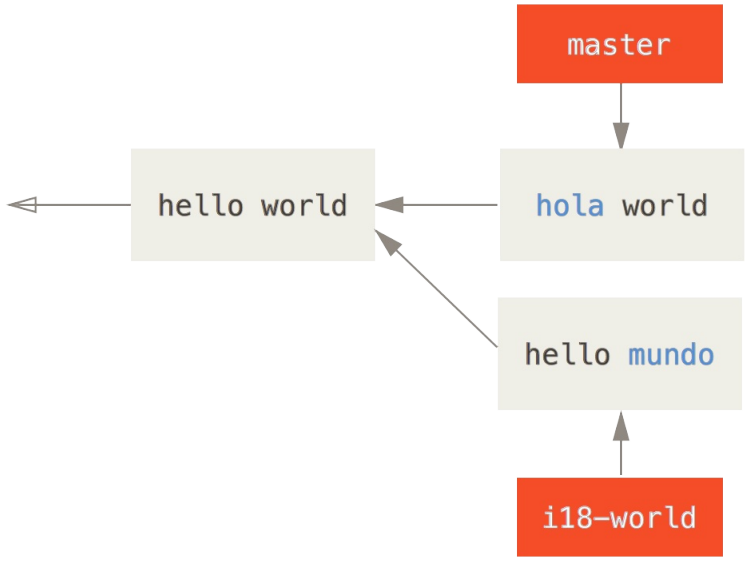


Figure 7-25.

当合并两个分支到一起时，我们将会得到一个合并冲突：

```
$ git merge i18n-world
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Recorded preimage for 'hello.rb'
Automatic merge failed; fix conflicts and then commit the result.
```

你会注意到那个新行 `Recorded preimage for FILE`。 除此之外它应该看起来就像一个普通的合并冲突。 在这个时候，`rerere` 可以告诉我们几件事。 和往常一样，在这个时候你可以运行 `git status` 来查看所有冲突的内容：

```
$ git status
# On branch master
# Unmerged paths:
#   (use "git reset HEAD <file>..." to unstage)
#   (use "git add <file>..." to mark resolution)
#
#       both modified:      hello.rb
#
```

然而，`git rerere` 也会通过 `git rerere status` 告诉你它记录的合并前状态。

```
$ git rerere status
hello.rb
```

并且 `git rerere diff` 将会显示解决方案的当前状态 - 开始解决前与解决后的样子。

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,11 @@
 #! /usr/bin/env ruby

 def hello
-<<<<<<<
-  puts 'hello mundo'
-=======
+<<<<<<< HEAD
+  puts 'hola world'
->>>>>>>
+=======
+  puts 'hello mundo'
+>>>>>>> i18n-world
 end
```

同样（这并不是真的与 `rerere` 有关系），可以使用 `ls-files -u` 来查看冲突文件的之前、左边与右边版本：

```
$ git ls-files -u
100644 39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1       hello.rb
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2       hello.rb
100644 54336ba847c3758ab604876419607e9443848474 3       hello.rb
```

现在可以通过改为 `puts 'hola mundo'` 来解决它，可以再次运行 `rerere diff` 命令来查看 rerere 将会记住的内容：

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,7 @@
 #! /usr/bin/env ruby

 def hello
-<<<<<<<
-  puts 'hello mundo'
-=======
-  puts 'hola world'
->>>>>>>
+  puts 'hola mundo'
 end
```

所以从本质上说，当 Git 看到一个 `hello.rb` 文件的一个块冲突中有 “hello mundo” 在一边与 “hola world” 在另一边，它会将其解决为 “hola mundo”。

现在我们可以将它标记为已解决并提交它：

```
$ git add hello.rb
$ git commit
Recorded resolution for 'hello.rb'.
[master 68e16e5] Merge branch 'i18n'
```

可以看到它 "Recorded resolution for FILE"。

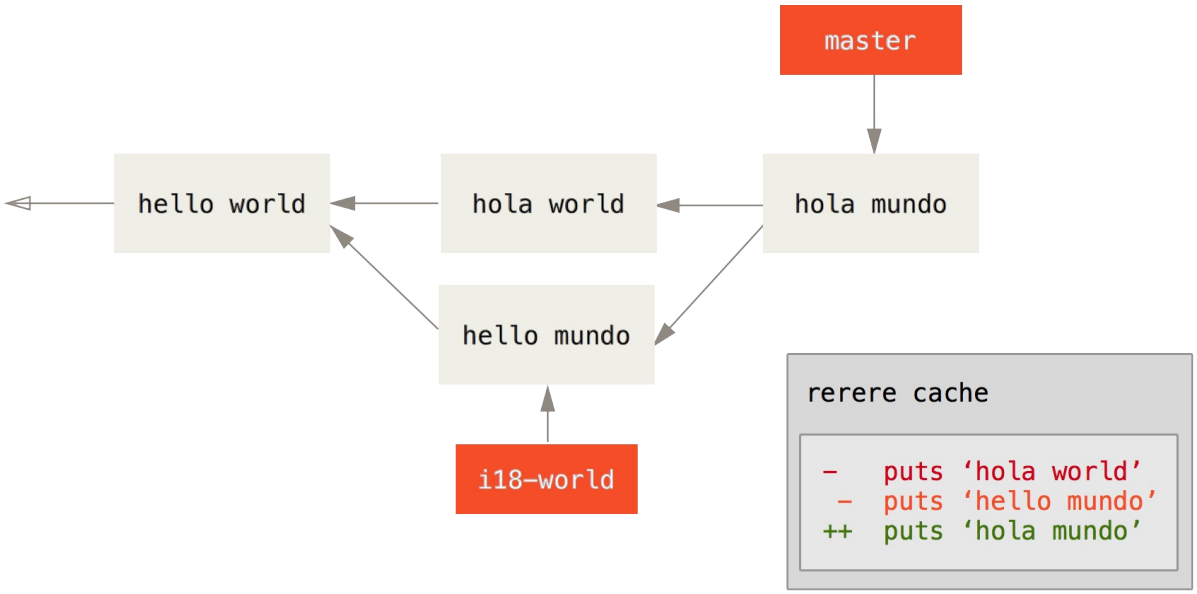


Figure 7-26.

现在，让我们撤消那个合并然后将它变基到 master 分支顶部来替代它。可以通过使用之前在 重置揭密 看到的 `reset` 来回滚分支。

```
$ git reset --hard HEAD^
HEAD is now at ad63f15 i18n the hello
```

我们的合并被撤消了。 现在让我们变基特性分支。

```
$ git checkout i18n-world
Switched to branch 'i18n-world'

$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: i18n one word
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
```

```
Failed to merge in the changes.
Patch failed at 0001 i18n one word
```

现在，正像我们期望的一样，得到了相同的合并冲突，但是看一下 `Resolved FILE using previous resolution` 这行。 如果我们看这个文件，会发现它已经被解决了，而且在它里面没有合并冲突标记。

```
$ cat hello.rb
#! /usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

同样， `git diff` 将会显示出它是如何自动地重新解决的：

```
$ git diff
diff --cc hello.rb
index a440db6,54336ba..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
 -  puts 'hello mundo'
++  puts 'hola mundo'
  end
```

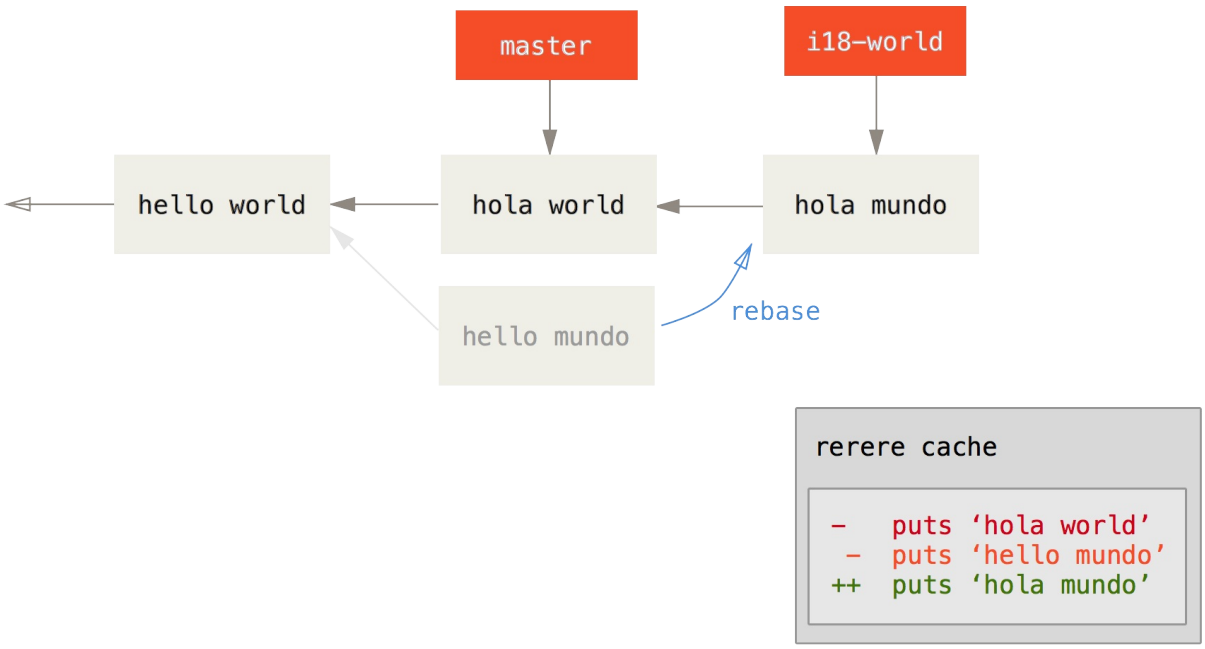


Figure 7-27.

也可以通过 `checkout` 命令重新恢复到冲突时候的文件状态：

```
$ git checkout --conflict=merge hello.rb
$ cat hello.rb
#! /usr/bin/env ruby

def hello
<<<<<<< ours
  puts 'hola world'
=======
  puts 'hello mundo'
>>>>>>> theirs
end
```

我们将会在 高级合并 中看到这个的一个例子。 然而现在，让我们通过运行 `rerere` 来重新解决它：

```
$ git rerere
Resolved 'hello.rb' using previous resolution.
$ cat hello.rb
#! /usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

我们通过 `rerere` 缓存的解决方案来自动重新解决了文件冲突。 现在可以添加并继续变基来完成它。

```
$ git add hello.rb
$ git rebase --continue
Applying: i18n one word
```

所以，如果做了很多次重新合并，或者想要一个特性分支始终与你的 master 分支保持最新但却不想要一大堆合并，或者经常变基，打开 `rerere` 功能可以帮助你的生活变得更美好。

7.10 使用 Git 调试

使用 Git 调试

Git 也提供了两个工具来辅助你调试项目中的问题。 由于 Git 被设计成适用于几乎所有类型的项目，这些工具是比较通用的，但它们可以在出现问题的时候帮助你找到 bug 或者错误。

文件标注

如果你在追踪代码中的一个 bug，并且想知道是什么时候以及为何会引入，文件标注通常是最好用的工具。 它展示了文件中每一行最后一次修改的提交。 所以，如果你在代码中看到一个有问题的方法，你可以使用 `git blame` 标注这个文件，查看这个方法每一行的最后修改时间以及是被谁修改的。 这个例子使用 `-L` 选项来限制输出范围在第12至22行：

```
$ git blame -L 12,22 simplegit.rb
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 12)  def show(tree = 'master')
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 13)   command("git show #{tree}")
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 14)  end
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 15)
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 16)  def log(tree = 'master')
79eaf55d (Scott Chacon  2008-04-06 10:15:08 -0700 17)   command("git log #{tree}")
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 18)  end
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 19)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20)  def blame(path)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21)   command("git blame #{path}")
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22)  end
```

请注意，第一个字段是最后一次修改该行的提交的部分 SHA-1 值。 接下来两个字段的值是从提交中提取出来的——作者的名字以及提交的时间——所以你就可以很轻易地找到是谁在什么时候修改了那一行。 接下来就是行号和文件内容。 注意一下 `^4832fe2` 这个提交的那些行，这些指的是这个文件第一次提交的那些行。 这个提交是这个文件第一次加入到这个项目时的提交，并且这些行从未被修改过。 这会带来小小的困惑，因为你已经至少看到三种 Git 使用 `^` 来修饰一个提交的 SHA-1 值的不同含义，但这里确实就是这个意思。

另一件比较酷的事情是 Git 不会显式地记录文件的重命名。 它会记录快照，然后在事后尝试计算出重命名的动作。 这其中有一个很有意思的特性就是你可以让 Git 找出所有的代码移动。 如果你在 `git blame` 后面加上一个 `-C` ，Git 会分析你正在标注的文件，并且尝试找出文件中从别的地方复制过来的代码片段的原始出处。 比如，你将 `GITServerHandler.m` 这个文件拆分为数个文件，其中一个文件是 `GITPackUpload.m` 。 对 `GITPackUpload.m` 执行带 `-C` 参数的blame命令，你就可以看到代码块的原始出处：

```
$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144)         //NSLog(@"GATHER COMMI
```

```
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 146)          NSString *parentSha;
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 147)          GITCommit *commit = [g
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 149)          //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 150)
56ef2caf GITServerHandler.m (Scott 2009-01-05 151)          if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152)                  [refDict setOb
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

这个功能很有用。 通常来说，你会认为复制代码过来的那个提交是最原始的提交，因为那是你第一次在这个文件中修改了这几行。 但 Git 会告诉你，你第一次写这几行代码的那个提交才是原始提交，即使这是在另外一个文件里写的。

二分查找

当你知道问题是在哪里引入的情况下文件标注可以帮助你查找问题。 如果你不知道哪里出了问题，并且自从上次可以正常运行到现在已经有数十个或者上百个提交，这个时候你可以使用 `git bisect` 来帮助查找。 `bisect` 命令会对你的提交历史进行二分查找来帮助你尽快找到是哪一个提交引入了问题。

假设你刚刚在线上环境部署了你的代码，接着收到一些 bug 反馈，但这些 bug 在你之前的开发环境里没有出现过，这让你百思不得其解。 你重新查看了你的代码，发现这个问题是可以被重现的，但是你不知道哪里出了问题。 你可以用二分法来找到这个问题。 首先执行 `git bisect start` 来启动，接着执行 `git bisect bad` 来告诉系统当前你所在的提交是有问题的。 然后你必须告诉 bisect 已知的最后一次正常状态是哪次提交，使用 `git bisect good [good_commit]` ：

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo
```

Git 发现在你标记为正常的提交(v1.0)和当前的错误版本之间有大约12次提交，于是 Git 检出中间的那个提交。 现在你可以执行测试，看看在这个提交下问题是不是还是存在。 如果还存在，说明问题是在这个提交之前引入的；如果问题不存在，说明问题是在这个提交之后引入的。 假设测试结果是没有问题的，你可以通过 `git bisect good` 来告诉 Git，然后继续寻找。

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

现在你在另一个提交上了，这个提交是刚刚那个测试通过的提交和有问题的提交的中点。 你再一次执行测试，发现这个提交下是有问题的，因此你可以通过 `git bisect bad` 告诉 Git：

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
```

```
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

这个提交是正常的，现在 Git 拥有的信息已经可以确定引入问题的位置在哪里。 它会告诉你第一个错误提交的 SHA-1 值并显示一些提交说明，以及哪些文件在那次提交里修改过，这样你可以找出引入 bug 的根源：

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date:   Tue Jan 27 14:48:32 2009 -0800

    secure this thing

:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcfc639b1a3814550e62d60b8e68a8e4 M  config
```

当你完成这些操作之后，你应该执行 `git bisect reset` 重置你的 HEAD 指针到最开始的位置，否则你会停留在一个很奇怪的状态：

```
$ git bisect reset
```

这是一个可以帮助你在几分钟内从数百个提交中找到 bug 的强大工具。 事实上，如果你有一个脚本在项目是正常的情况下返回 0，在不正常的情况下返回非 0，你可以使 `git bisect` 自动化这些操作。 首先，你设定好项目正常以及不正常所在提交的二分查找范围。 你可以通过 `bisect start` 命令的参数来设定这两个提交，第一个参数是项目不正常的提交，第二个参数是项目正常的提交：

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

Git 会自动在每个被检出的提交里执行 `test-error.sh` 直到找到第一个项目不正常的提交。 你也可以执行 `make` 或者 `make tests` 或者其他东西来进行自动化测试。

7.11 子模块

子模块

有种情况我们经常会遇到：某个工作中的项目需要包含并使用另一个项目。 也许是第三方库，或者你独立开发的，用于多个父项目的库。 现在问题来了：你想要把它们当做两个独立的项目，同时又想在一个项目中使用另一个。

我们举一个例子。 假设你正在开发一个网站然后创建了 Atom 订阅。 你决定使用一个库，而不是写自己的 Atom 生成代码。 你可能不得不通过 CPAN 安装或 Ruby gem 来包含共享库中的代码，或者将源代码直接拷贝到自己的项目中。 如果将这个库包含进来，那么无论用何种方式都很难定制它，部署则更加困难，因为你必须确保每一个客户端都包含该库。 如果将代码复制到自己的项目中，那么你做的任何自定义修改都会使合并上游的改动变得困难。

Git 通过子模块来解决这个问题。 子模块允许你将一个 Git 仓库作为另一个 Git 仓库的子目录。 它能让你将另一个仓库克隆到自己的项目中，同时还保持提交的独立。

开始使用子模块

我们将要演示如何在一个被分成一个主项目与几个子项目的项目上开发。

我们首先将一个已存在的 Git 仓库添加为正在工作的仓库的子模块。 你可以通过在 `git submodule add` 命令后面加上想要跟踪的项目 URL 来添加新的子模块。 在本例中，我们将会添加一个名为 “DbConnector” 的库。

```
$ git submodule add https://github.com/chaconinc/DbConnector
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

默认情况下，子模块会将子项目放到一个与仓库同名的目录中，本例中是 “DbConnector”。 如果你想要放到其他地方，那么可以在命令结尾添加一个不同的路径。

如果这时运行 `git status`，你会注意到几件事。

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   .gitmodules
```

```
        new file:   DbConnector
```

首先应当注意到新的 `.gitmodules` 文件。 该置文件保存了项目 URL 与已经拉取的本地目录之间的映射：

```
$ cat .gitmodules
[submodule "DbConnector"]
        path = DbConnector
        url = https://github.com/chaconinc/DbConnector
```

如果有多个子模块，该文件中就会有多条记录。 要重点注意的是，该文件也像 `.gitignore` 文件一样受到（通过）版本控制。 它会和该项目的其他部分一同被拉取推送。 这就是克隆该项目的人知道去哪获得子模块的原因。

> NOTE
>
> 由于 .gitmodules 文件中的 URL 是人们首先尝试克隆/拉取的地方，因此请尽可能确保你使用的URL大家都能访问。 例如，若你要使用的推送 URL 与他人的拉取 URL 不同，那么请使用他人能访问到的 URL。 你也可以根据自己的需要，通过在本地执行 `git config submodule.DbConnector.url <私有URL>` 来覆盖这个选项的值。 如果可行的话，一个相对路径会很有帮助。

在 `git status` 输出中列出的另一个是项目文件夹记录。 如果你运行 `git diff`，会看到类似下面的信息：

```
$ git diff --cached DbConnector
diff --git a/DbConnector b/DbConnector
new file mode 160000
index 0000000..c3f01dc
--- /dev/null
+++ b/DbConnector
@@ -0,0 +1 @@
+Subproject commit c3f01dc8862123d317dd46284b05b6892c7b29bc
```

虽然 `DbConnector` 是工作目录中的一个子目录，但 Git 还是会将它视作一个子模块。当你不在那个目录中时，Git 并不会跟踪它的内容， 而是将它看作该仓库中的一个特殊提交。

如果你想看到更漂亮的差异输出，可以给 `git diff` 传递 `--submodule` 选项。

```
$ git diff --cached --submodule
diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000..71fc376
--- /dev/null
```

```
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "DbConnector"]
+       path = DbConnector
+       url = https://github.com/chaconinc/DbConnector
Submodule DbConnector 0000000...c3f01dc (new submodule)
```

当你提交时，会看到类似下面的信息：

```
$ git commit -am 'added DbConnector module'
[master fb9093c] added DbConnector module
 2 files changed, 4 insertions(+)
 create mode 100644 .gitmodules
 create mode 160000 DbConnector
```

注意 DbConnector 记录的 `160000` 模式。 这是 Git 中的一种特殊模式，它本质上意味着你是将一次提交记作一项目录记录的，而非将它记录成一个子目录或者一个文件。

克隆含有子模块的项目

接下来我们将会克隆一个含有子模块的项目。 当你在克隆这样的项目时，默认会包含该子模块目录，但其中还没有任何文件：

```
$ git clone https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
$ cd MainProject
$ ls -la
total 16
drwxr-xr-x   9 schacon  staff  306 Sep 17 15:21 .
drwxr-xr-x   7 schacon  staff  238 Sep 17 15:21 ..
drwxr-xr-x  13 schacon  staff  442 Sep 17 15:21 .git
-rw-r--r--   1 schacon  staff   92 Sep 17 15:21 .gitmodules
drwxr-xr-x   2 schacon  staff   68 Sep 17 15:21 DbConnector
-rw-r--r--   1 schacon  staff  756 Sep 17 15:21 Makefile
drwxr-xr-x   3 schacon  staff  102 Sep 17 15:21 includes
drwxr-xr-x   4 schacon  staff  136 Sep 17 15:21 scripts
drwxr-xr-x   4 schacon  staff  136 Sep 17 15:21 src
$ cd DbConnector/
$ ls
$
```

其中有 `DbConnector` 目录，不过是空的。 你必须运行两个命令： `git submodule init` 用来初始化本地配置文件，而 `git submodule update` 则从该项目中抓取所有数据并检出父项目中列出的合适的提交。

```
$ git submodule init
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path 'DbConnector'
```

```
$ git submodule update
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

现在 `DbConnector` 子目录是处在和之前提交时相同的状态了。

不过还有更简单一点的方式。 如果给 `git clone` 命令传递 `--recursive` 选项，它就会自动初始化并更新仓库中的每一个子模块。

```
$ git clone --recursive https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path 'DbConnector'
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

在包含子模块的项目上工作

现在我们有一份包含子模块的项目副本，我们将会同时在主项目和子模块项目上与队员协作。

拉取上游修改

在项目中使用子模块的最简模型，就是只使用子项目并不时地获取更新，而并不在你的检出中进行任何更改。 我们来看一个简单的例子。

如果想要在子模块中查看新工作，可以进入到目录中运行 `git fetch` 与 `git merge` ，合并上游分支来更新本地代码。

```
$ git fetch
From https://github.com/chaconinc/DbConnector
   c3f01dc..d0354fc  master     -> origin/master
$ git merge origin/master
Updating c3f01dc..d0354fc
Fast-forward
 scripts/connect.sh | 1 +
 src/db.c           | 1 +
 2 files changed, 2 insertions(+)
```

如果你现在返回到主项目并运行 `git diff --submodule` ，就会看到子模块被更新的同时获得了一个

包含新添加提交的列表。 如果你不想每次运行 `git diff` 时都输入 `--submodle`，那么可以将 `diff.submodule` 设置为 “log” 来将其作为默认行为。

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
  > more efficient db routine
  > better connection routine
```

如果在此时提交，那么你会将子模块锁定为其他人更新时的新代码。

如果你不想在子目录中手动抓取与合并，那么还有种更容易的方式。 运行 `git submodule update --remote`，Git 将会进入子模块然后抓取并更新。

```
$ git submodule update --remote DbConnector
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   3f19983..d0354fc  master     -> origin/master
Submodule path 'DbConnector': checked out 'd0354fc054692d3906c85c3af05ddce39a1c0644'
```

此命令默认会假定你想要更新并检出子模块仓库的 `master` 分支。 不过你也可以设置为想要的其他分支。 例如，你想要 DbConnector 子模块跟踪仓库的 “stable” 分支，那么既可以在 `.gitmodules` 文件中设置（这样其他人也可以跟踪它），也可以只在本地的 `.git/config` 文件中设置。 让我们在 `.gitmodules` 文件中设置它：

```
$ git config -f .gitmodules submodule.DbConnector.branch stable

$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   27cf5d3..c87d55d  stable -> origin/stable
Submodule path 'DbConnector': checked out 'c87d55d4c6d4b05ee34fbc8cb6f7bf4585ae6687'
```

如果不用 `-f .gitmodules` 选项，那么它只会为你做修改。但是在仓库中保留跟踪信息更有意义一些，因为其他人也可以得到同样的效果。

这时我们运行 `git status`，Git 会显示子模块中有 “新提交”。

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
```

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

  modified:   .gitmodules
  modified:   DbConnector (new commits)

no changes added to commit (use "git add" and/or "git commit -a")
```

如果你设置了配置选项 `status.submodulesummary`，Git 也会显示你的子模块的更改摘要：

```
$ git config status.submodulesummary 1

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   .gitmodules
        modified:   DbConnector (new commits)

Submodules changed but not updated:

* DbConnector c3f01dc...c87d55d (4):
  > catch non-null terminated lines
```

这时如果运行 `git diff`，可以看到我们修改了 .gitmodules 文件，同时还有几个已拉取的提交需要提交到我们自己的子模块项目中。

```
$ git diff
diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
        path = DbConnector
        url = https://github.com/chaconinc/DbConnector
+       branch = stable
 Submodule DbConnector c3f01dc..c87d55d:
  > catch non-null terminated lines
  > more robust error handling
  > more efficient db routine
  > better connection routine
```

这非常有趣，因为我们可以直接看到将要提交到子模块中的提交日志。 提交之后，你也可以运行 `git log -p` 查看这个信息。

```
$ git log -p --submodule
commit 0a24cfc121a8a3c118e0105ae4ae4c00281cf7ae
Author: Scott Chacon <schacon@gmail.com>
```

```
Date:   Wed Sep 17 16:37:02 2014 +0200

    updating DbConnector for bug fixes

diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
        path = DbConnector
        url = https://github.com/chaconinc/DbConnector
+       branch = stable
Submodule DbConnector c3f01dc..c87d55d:
  > catch non-null terminated lines
  > more robust error handling
  > more efficient db routine
  > better connection routine
```

当运行 `git submodule update --remote` 时，Git 默认会尝试更新所有子模块，所以如果有很多子模块的话，你可以传递想要更新的子模块的名字。

在子模块上工作

你很有可能正在使用子模块，因为你确实想在子模块中编写代码的同时，还想在主项目上编写代码（或者跨子模块工作）。 否则你大概只能用简单的依赖管理系统（如 Maven 或 Rubygems）来替代了。

现在我们将通过一个例子来演示如何在子模块与主项目中同时做修改，以及如何同时提交与发布那些修改。

到目前为止，当我们运行 `git submodule update` 从子模块仓库中抓取修改时，Git 将会获得这些改动并更新子目录中的文件，但是会将子仓库留在一个称作 “游离的 HEAD” 的状态。 这意味着没有本地工作分支（例如 “master” ）跟踪改动。 所以你做的任何改动都不会被跟踪。

为了将子模块设置得更容易进入并修改，你需要做两件事。 首先，进入每个子模块并检出其相应的工作分支。 接着，若你做了更改就需要告诉 Git 它该做什么，然后运行 `git submodule update --remote` 来从上游拉取新工作。 你可以选择将它们合并到你的本地工作中，也可以尝试将你的工作变基到新的更改上。

首先，让我们进入子模块目录然后检出一个分支。

```
$ git checkout stable
Switched to branch 'stable'
```

然后尝试用 “merge” 选项。 为了手动指定它，我们只需给 `update` 添加 `--merge` 选项即可。 这时我们将会看到服务器上的这个子模块有一个改动并且它被合并了进来。

```
$ git submodule update --remote --merge
remote: Counting objects: 4, done.
```

```
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   c87d55d..92c7337  stable     -> origin/stable
Updating c87d55d..92c7337
Fast-forward
 src/main.c | 1 +
 1 file changed, 1 insertion(+)
Submodule path 'DbConnector': merged in '92c7337b30ef9e0893e758dac2459d07362ab5ea'
```

如果我们进入 DbConnector 目录，可以发现新的改动已经合并入本地 `stable` 分支。 现在让我们看看当我们对库做一些本地的改动而同时其他人推送另外一个修改到上游时会发生什么。

```
$ cd DbConnector/
$ vim src/db.c
$ git commit -am 'unicode support'
[stable f906e16] unicode support
 1 file changed, 1 insertion(+)
```

如果我们现在更新子模块，就会看到当我们在本地做了更改时上游也有一个改动，我们需要将它并入本地。

```
$ git submodule update --remote --rebase
First, rewinding head to replay your work on top of it...
Applying: unicode support
Submodule path 'DbConnector': rebased into '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

如果你忘记 `--rebase` 或 `--merge`，Git 会将子模块更新为服务器上的状态。并且会将项目重置为一个游离的 HEAD 状态。

```
$ git submodule update --remote
Submodule path 'DbConnector': checked out '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

即便这真的发生了也不要紧，你只需回到目录中再次检出你的分支（即还包含着你的工作的分支）然后手动地合并或变基 `origin/stable`（或任何一个你想要的远程分支）就行了。

如果你没有提交子模块的改动，那么运行一个子模块更新也不会出现问题，此时 Git 会只抓取更改而并不会覆盖子模块目录中未保存的工作。

```
$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 4 (delta 0)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   5d60ef9..c75e92a  stable     -> origin/stable
error: Your local changes to the following files would be overwritten by checkout:
```

```
        scripts/setup.sh
Please, commit your changes or stash them before you can switch branches.
Aborting
Unable to checkout 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path 'DbConnector'
```

如果你做了一些与上游改动冲突的改动，当运行更新时 Git 会让你知道。

```
$ git submodule update --remote --merge
Auto-merging scripts/setup.sh
CONFLICT (content): Merge conflict in scripts/setup.sh
Recorded preimage for 'scripts/setup.sh'
Automatic merge failed; fix conflicts and then commit the result.
Unable to merge 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path 'DbConnector'
```

你可以进入子模块目录中然后就像平时那样修复冲突。

发布子模块改动

现在我们的子模块目录中有一些改动。 其中有一些是我们通过更新从上游引入的，而另一些是本地生成的，由于我们还没有推送它们，所以对任何其他人都不可用。

```
$ git diff
Submodule DbConnector c87d55d..82d2ad3:
  > Merge from origin/stable
  > updated setup script
  > unicode support
  > remove unnecessary method
  > add new option for conn pooling
```

如果我们在主项目中提交并推送但并不推送子模块上的改动，其他尝试检出我们修改的人会遇到麻烦，因为他们无法得到依赖的子模块改动。 那些改动只存在于我们本地的拷贝中。

为了确保这不会发生，你可以让 Git 在推送到主项目前检查所有子模块是否已推送。 `git push` 命令接受可以设置为 “check” 或 “on-demand” 的 `--recurse-submodules` 参数。 如果任何提交的子模块改动没有推送那么 “check” 选项会直接使 `push` 操作失败。

```
$ git push --recurse-submodules=check
The following submodule paths contain changes that can
not be found on any remote:
  DbConnector

Please try

        git push --recurse-submodules=on-demand

or cd to the path and use

        git push

to push them to a remote.
```

如你所见，它也给我们了一些有用的建议，指导接下来该如何做。 最简单的选项是进入每一个子模块中然后手动推送到远程仓库，确保它们能被外部访问到，之后再次尝试这次推送。

另一个选项是使用 “on-demand” 值，它会尝试为你这样做。

```
$ git push --recurse-submodules=on-demand
Pushing submodule 'DbConnector'
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 3), reused 0 (delta 0)
To https://github.com/chaconinc/DbConnector
   c75e92a..82d2ad3  stable -> stable
Counting objects: 2, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 266 bytes | 0 bytes/s, done.
Total 2 (delta 1), reused 0 (delta 0)
To https://github.com/chaconinc/MainProject
   3d6d338..9a377d1  master -> master
```

如你所见，Git 进入到 DbConnector 模块中然后在推送主项目前推送了它。 如果那个子模块因为某些原因推送失败，主项目也会推送失败。

合并子模块改动

如果你其他人同时改动了一个子模块引用，那么可能会遇到一些问题。 也就是说，如果子模块的历史已经分叉并且在父项目中分别提交到了分叉的分支上，那么你需要做一些工作来修复它。

如果一个提交是另一个的直接祖先（一个快进式合并），那么 Git 会简单地选择之后的提交来合并，这样没什么问题。

不过，Git 甚至不会尝试去进行一次简单的合并。 如果子模块提交已经分叉且需要合并，那你会得到类似下面的信息：

```
$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1)
Unpacking objects: 100% (2/2), done.
From https://github.com/chaconinc/MainProject
   9a377d1..eb974f8  master     -> origin/master
Fetching submodule DbConnector
warning: Failed to merge submodule DbConnector (merge following commits not found)
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

所以本质上 Git 在这里指出了子模块历史中的两个分支记录点已经分叉并且需要合并。 它将其解释为 “merge following commits not found”（未找到接下来需要合并的提交），虽然这有点令人困惑，不

过之后我们会解释为什么是这样。

为了解决这个问题，你需要弄清楚子模块应该处于哪种状态。 奇怪的是，Git 并不会给你多少能帮你摆脱困境的信息，甚至连两边提交历史中的 SHA-1 值都没有。 幸运的是，这很容易解决。 如果你运行 `git diff`，就会得到试图合并的两个分支中记录的提交的 SHA-1 值。

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
```

所以，在本例中，`eb41d76` 是我们的子模块中大家共有的提交，而 `c771610` 是上游拥有的提交。 如果我们进入子模块目录中，它应该已经在 `eb41d76` 上了，因为合并没有动过它。 如果不是的话，无论什么原因，你都可以简单地创建并检出一个指向它的分支。

来自另一边的提交的 SHA-1 值比较重要。 它是需要你来合并解决的。 你可以尝试直接通过 SHA-1 合并，也可以为它创建一个分支然后尝试合并。 我们建议后者，哪怕只是为了一个更漂亮的合并提交信息。

所以，我们将会进入子模块目录，基于 `git diff` 的第二个 SHA 创建一个分支然后手动合并。

```
$ cd DbConnector

$ git rev-parse HEAD
eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610
(DbConnector) $ git merge try-merge
Auto-merging src/main.c
CONFLICT (content): Merge conflict in src/main.c
Recorded preimage for 'src/main.c'
Automatic merge failed; fix conflicts and then commit the result.
```

我们在这儿得到了一个真正的合并冲突，所以如果想要解决并提交它，那么只需简单地通过结果来更新主项目。

```
$ vim src/main.c
$ git add src/main.c
$ git commit -am 'merged our changes'
Recorded resolution for 'src/main.c'.
[master 9fd905e] merged our changes

$ cd ..
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
@@@ -1,1 -1,1 +1,1 @@@
- Subproject commit eb41d764bccf88be77aced643c13a7fa86714135
```

```
  -Subproject commit c77161012afbbe1f58b5053316ead08f4b7e6d1d
++Subproject commit 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
$ git add DbConnector

$ git commit -m "Merge Tom's Changes"
[master 10d2c60] Merge Tom's Changes
```

❶ 首先解决冲突

❷ 然后返回到主项目目录中

❸ 再次检查 SHA-1 值

❹ 解决冲突的子模块记录

❺ 提交我们的合并

这可能会让你有点儿困惑，但它确实不难。

有趣的是，Git 还能处理另一种情况。 如果子模块目录中存在着这样一个合并提交，它的历史中包含了的两边的提交，那么 Git 会建议你将它作为一个可行的解决方案。 它看到有人在子模块项目的某一点上合并了包含这两次提交的分支，所以你可能想要那个。

这就是为什么前面的错误信息是 “merge following commits not found” ，因为它不能 这样做。 它让人困惑是因为谁能想到它会尝试这样做？

如果它找到了一个可以接受的合并提交，你会看到类似下面的信息：

```
$ git merge origin/master
warning: Failed to merge submodule DbConnector (not fast-forward)
Found a possible merge resolution for the submodule:
 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a: > merged our changes
If this is correct simply add it to the index for example
by using:

  git update-index --cacheinfo 160000 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a "DbConnector"

which will accept this suggestion.
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

它会建议你更新索引，就像你运行了 `git add` 那样，这样会清除冲突然后提交。不过你可能不应该这样做。你可以轻松地进入子模块目录，查看差异是什么，快进到这次提交，恰当地测试，然后提交它。

```
$ cd DbConnector/
$ git merge 9fd905e
Updating eb41d76..9fd905e
Fast-forward
```

```
$ cd ..
$ git add DbConnector
$ git commit -am 'Fast forwarded to a common submodule child'
```

这些命令完成了同一件事，但是通过这种方式你至少可以验证工作是否有效，以及当你在完成时可以确保子模块目录中有你的代码。

子模块技巧

你可以做几件事情来让用子模块工作轻松一点儿。

子模块遍历

有一个 `foreach` 子模块命令，它能在每一个子模块中运行任意命令。 如果项目中包含了大量子模块，这会非常有用。

例如，假设我们想要开始开发一项新功能或者修复一些错误，并且需要在几个子模块内工作。 我们可以轻松地保存所有子模块的工作进度。

```
$ git submodule foreach 'git stash'
Entering 'CryptoLibrary'
No local changes to save
Entering 'DbConnector'
Saved working directory and index state WIP on stable: 82d2ad3 Merge from origin/stable
HEAD is now at 82d2ad3 Merge from origin/stable
```

然后我们可以创建一个新分支，并将所有子模块都切换过去。

```
$ git submodule foreach 'git checkout -b featureA'
Entering 'CryptoLibrary'
Switched to a new branch 'featureA'
Entering 'DbConnector'
Switched to a new branch 'featureA'
```

你应该明白。 能够生成一个主项目与所有子项目的改动的统一差异是非常有用的。

```
$ git diff; git submodule foreach 'git diff'
Submodule DbConnector contains modified content
diff --git a/src/main.c b/src/main.c
index 210f1ae..1f0acdc 100644
--- a/src/main.c
+++ b/src/main.c
@@ -245,6 +245,8 @@ static int handle_alias(int *argcp, const char ***argv)

      commit_pager_choice();

+     url = url_decode(url_orig);
+
      /* build alias_argv */
      alias_argv = xmalloc(sizeof(*alias_argv) * (argc + 1));
      alias_argv[0] = alias_string + 1;
Entering 'DbConnector'
```

```
diff --git a/src/db.c b/src/db.c
index 1aaefb6..5297645 100644
--- a/src/db.c
+++ b/src/db.c
@@ -93,6 +93,11 @@ char *url_decode_mem(const char *url, int len)
        return url_decode_internal(&url, len, NULL, &out, 0);
 }

+char *url_decode(const char *url)
+{
+       return url_decode_mem(url, strlen(url));
+}
+
 char *url_decode_parameter_name(const char **query)
 {
        struct strbuf out = STRBUF_INIT;
```

在这里，我们看到子模块中定义了一个函数并在主项目中调用了它。 这明显是个简化了的例子，但是希望它能让你明白这种方法的用处。

有用的别名

你可能想为其中一些命令设置别名，因为它们可能会非常长而你又不能设置选项作为它们的默认选项。 我们在 Git 别名 介绍了设置 Git 别名，但是如果你计划在 Git 中大量使用子模块的话，这里有一些例子。

```
$ git config alias.sdiff '!'"git diff && git submodule foreach 'git diff'"
$ git config alias.spush 'push --recurse-submodules=on-demand'
$ git config alias.supdate 'submodule update --remote --merge'
```

这样当你想要更新子模块时可以简单地运行 `git supdate` ，或 `git spush` 检查子模块依赖后推送。

子模块的问题

然而使用子模块还是有一些小问题。

例如在有子模块的项目中切换分支可能会造成麻烦。 如果你创建一个新分支，在其中添加一个子模块，之后切换到没有该子模块的分支上时，你仍然会有一个还未跟踪的子模块目录。

```
$ git checkout -b add-crypto
Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
...

$ git commit -am 'adding crypto library'
[add-crypto 4445836] adding crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
```

```
Your branch is up-to-date with 'origin/master'.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        CryptoLibrary/

nothing added to commit but untracked files present (use "git add" to track)
```

移除那个目录并不困难，但是有一个目录在那儿会让人有一点困惑。 如果你移除它然后切换回有那个子模块的分支，需要运行 `submodule update --init` 来重新建立和填充。

```
$ git clean -fdx
Removing CryptoLibrary/

$ git checkout add-crypto
Switched to branch 'add-crypto'

$ ls CryptoLibrary/

$ git submodule update --init
Submodule path 'CryptoLibrary': checked out 'b8dda6aa182ea4464f3f3264b11e0268545172af'

$ ls CryptoLibrary/
Makefile        includes        scripts         src
```

再说一遍，这真的不难，只是会让人有点儿困惑。

另一个主要的告诫是许多人遇到了将子目录转换为子模块的问题。 如果你在项目中已经跟踪了一些文件，然后想要将它们移动到一个子模块中，那么请务必小心，否则 Git 会对你发脾气。 假设项目内有一些文件在子目录中，你想要将其转换为一个子模块。 如果删除子目录然后运行 `submodule add`，Git 会朝你大喊：

```
$ rm -Rf CryptoLibrary/
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

你必须要先取消暂存 `CryptoLibrary` 目录。 然后才可以添加子模块：

```
$ git rm -r CryptoLibrary
$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

现在假设你在一个分支下做了这样的工作。 如果尝试切换回的分支中那些文件还在子目录而非子模块中时 - 你会得到这个错误：

```
$ git checkout master
error: The following untracked working tree files would be overwritten by checkout:
  CryptoLibrary/Makefile
  CryptoLibrary/includes/crypto.h
  ...
Please move or remove them before you can switch branches.
Aborting
```

你可以通过 `check -f` 来强制切换，但是要小心，如果其中还有未保存的修改，这个命令会把它们覆盖掉。

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
```

当你切换回来之后，因为某些原因你得到了一个空的 `CryptoLibrary` 目录，并且 `git submodule update` 也无法修复它。 你需要进入到子模块目录中运行 `git checkout .` 来找回所有的文件。 你也可以通过 `submodule foreach` 脚本来为多个子模块运行它。

要特别注意的是，近来子模块会将它们的所有 Git 数据保存在顶级项目的 `.git` 目录中，所以不像旧版本的 Git，摧毁一个子模块目录并不会丢失任何提交或分支。

拥有了这些工具，使用子模块会成为可以在几个相关但却分离的项目上同时开发的相当简单有效的方法。

7.12 打包

打包

虽然我们已经了解了网络传输 Git 数据的常用方法（如 HTTP，SSH 等），但还有另外一种不太常见却又十分有用的方式。

Git 可以将它的数据 “打包” 到一个文件中。 这在许多场景中都很有用。 有可能你的网络中断了，但你又希望将你的提交传给你的合作者们。 可能你不在办公网中并且出于安全考虑没有给你接入内网的权限。 可能你的无线、有线网卡坏掉了。 可能你现在没有共享服务器的权限，你又希望通过邮件将更新发送给别人，却不希望通过 `format-patch` 的方式传输 40 个提交。

这些情况下 `git bundle` 就会很有用。 `bundle` 命令会将 `git push` 命令所传输的所有内容打包成一个二进制文件，你可以将这个文件通过邮件或者闪存传给其他人，然后解包到其他的仓库中。

来看看一个简单的例子。 假设你有一个包含两个提交的仓库：

```
$ git log
commit 9a466c572fe88b195efd356c3f2bbeccdb504102
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:10 2010 -0800

    second commit

commit b1ec3248f39900d2a406049d762aa68e9641be25
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:01 2010 -0800

    first commit
```

如果你想把这个仓库发送给其他人但你没有其他仓库的权限，或者就是懒得新建一个仓库，你就可以用 `git bundle create` 命令来打包。

```
$ git bundle create repo.bundle HEAD master
Counting objects: 6, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 441 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
```

然后你就会有一个名为 `repo.bundle` 的文件，该文件包含了所有重建该仓库 `master` 分支所需的数据。 在使用 `bundle` 命令时，你需要列出所有你希望打包的引用或者提交的区间。 如果你希望这个仓库可以在别处被克隆，你应该像例子中那样增加一个 HEAD 引用。

你可以将这个 `repo.bundle` 文件通过邮件或者U盘传给别人。

另一方面，假设别人传给你一个 `repo.bundle` 文件并希望你在这个项目上工作。 你可以从这个二进制文件中克隆出一个目录，就像从一个 URL 克隆一样。

```
$ git clone repo.bundle repo
Initialized empty Git repository in /private/tmp/bundle/repo/.git/
$ cd repo
$ git log --oneline
9a466c5 second commit
b1ec324 first commit
```

如果你在打包时没有包含 HEAD 引用，你还需要在命令后指定一个 `-b master` 或者其他被引入的分支，否则 Git 不知道应该检出哪一个分支。

现在假设你提交了 3 个修订，并且要用邮件或者U盘将新的提交放在一个包里传回去。

```
$ git log --oneline
71b84da last commit - second repo
c99cf5b fourth commit - second repo
7011d3d third commit - second repo
9a466c5 second commit
b1ec324 first commit
```

首先我们需要确认我们希望被打包的提交区间。 和网络协议不太一样，网络协议会自动计算出所需传输的最小数据集，而我们需要手动计算。 当然你可以像上面那样将整个仓库打包，但最好仅仅打包变更的部分 —— 就是我们刚刚在本地做的 3 个提交。

为了实现这个目标，你需要计算出差别。 就像我们在 提交区间 介绍的，你有很多种方式去指明一个提交区间。 我们可以使用 `origin/master..master` 或者 `master ^origin/master` 之类的方法来获取那 3 个在我们的 master 分支而不在原始仓库中的提交。 你可以用 `log` 命令来测试。

```
$ git log --oneline master ^origin/master
71b84da last commit - second repo
c99cf5b fourth commit - second repo
7011d3d third commit - second repo
```

这样就获取到我们希望被打包的提交列表，让我们将这些提交打包。 我们可以用 `git bundle create` 命令，加上我们想用的文件名，以及要打包的提交区间。

```
$ git bundle create commits.bundle master ^9a466c5
Counting objects: 11, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (9/9), 775 bytes, done.
Total 9 (delta 0), reused 0 (delta 0)
```

现在在我们的目录下会有一个 `commits.bundle` 文件。 如果我们把这个文件发送给我们的合作者，她

可以将这个文件导入到原始的仓库中，即使在这期间已经有其他的工作提交到这个仓库中。

当她拿到这个包时，她可以在导入到仓库之前查看这个包里包含了什么内容。 `bundle verify` 命令可以检查这个文件是否是一个合法的 Git 包，是否拥有共同的祖先来导入。

```
$ git bundle verify ../commits.bundle
The bundle contains 1 ref
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
The bundle requires these 1 ref
9a466c572fe88b195efd356c3f2bbeccdb504102 second commit
../commits.bundle is okay
```

如果打包工具仅仅把最后两个提交打包，而不是三个，原始的仓库是无法导入这个包的，因为这个包缺失了必要的提交记录。这时候 `verify` 的输出类似：

```
$ git bundle verify ../commits-bad.bundle
error: Repository lacks these prerequisite commits:
error: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 third commit - second repo
```

而我们的第一个包是合法的，所以我们可以从这个包里提取出提交。 如果你想查看这边包里可以导入哪些分支，同样有一个命令可以列出这些顶端：

```
$ git bundle list-heads ../commits.bundle
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

`verify` 子命令同样可以告诉你有哪些顶端。 该功能的目的是查看哪些是可以被拉入的，所以你可以使用 `fetch` 或者 `pull` 命令从包中导入提交。 这里我们要从包中取出 `master` 分支到我们仓库中的 other-master 分支：

```
$ git fetch ../commits.bundle master:other-master
From ../commits.bundle
 * [new branch]      master     -> other-master
```

可以看到我们已经将提交导入到 other-master 分支，以及在这期间我们自己在 master 分支上的提交。

```
$ git log --oneline --decorate --graph --all
* 8255d41 (HEAD, master) third commit - first repo
| * 71b84da (other-master) last commit - second repo
| * c99cf5b fourth commit - second repo
| * 7011d3d third commit - second repo
|/
* 9a466c5 second commit
* b1ec324 first commit
```

因此，当你在没有合适的网络或者可共享仓库的情况下， `git bundle` 很适合用于共享或者网络类型的

操作。

7.13 替换

替换

Git 对象是不可改变的，但它提供一种有趣的方式来用其他对象假装替换数据库中的 Git 对象。

`replace` 命令可以让你在 Git 中指定一个对象并可以声称“每次你遇到这个 Git 对象时，假装它是其他的东西”。 在你用一个不同的提交替换历史中的一个提交时，这会非常有用。

例如，你有一个大型的代码历史并想把自己的仓库分成一个短的历史和一个更大更长久的历史，短历史供新的开发者使用，后者给喜欢数据挖掘的人使用。 你可以通过用新仓库中最早的提交 `replace` 老仓库中最新的提交来连接历史，这种方式可以把一条历史移植到其他历史上。 这意味着你不用在新历史中真正替换每一个提交（因为历史来源会影响 SHA 的值），你可以加入他们。

让我们来试试吧。 首先获取一个已经存在的仓库，并将其分成两个仓库，一个是最近的仓库，一个是历史版本的仓库，然后我们将看到如何在不更改仓库 SHA 值的情况下通过 `replace` 命令来合并他们。

我们将使用一个拥有 5 个提交的简单仓库：

```
$ git log --oneline
ef989d8 fifth commit
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

我们想将其分成拆分成两条历史。 第一个到第四个提交的作为第一个历史版本。 第四、第五个提交的作为最近的第二个历史版本。

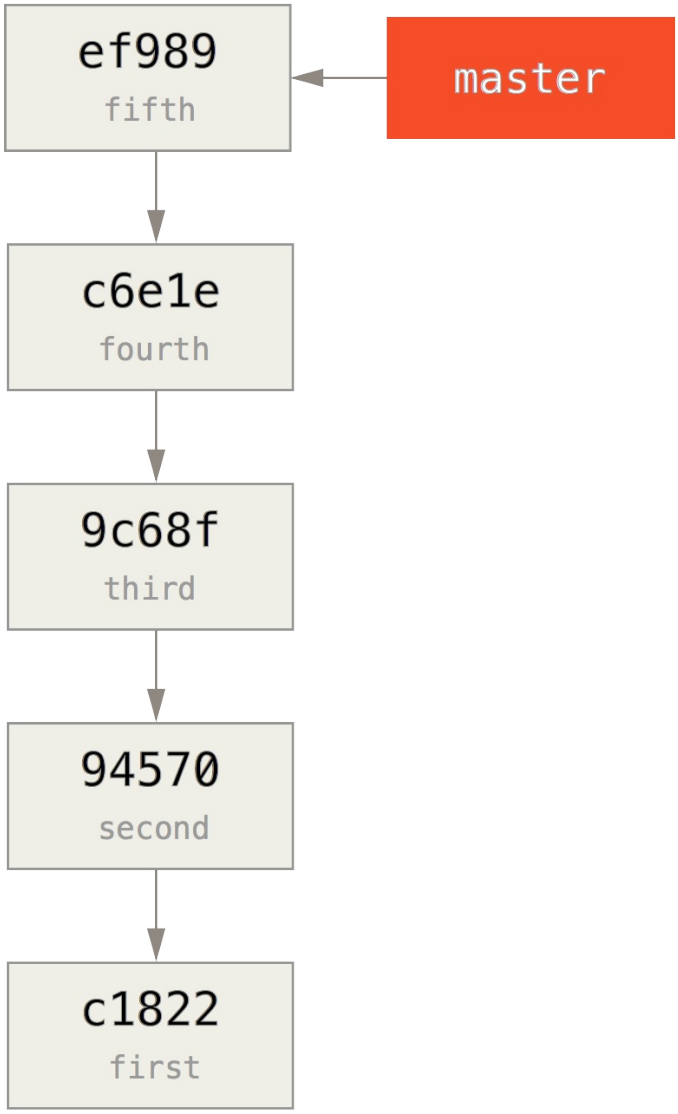


Figure 7-28.

创建历史版本的历史很容易，我们可以只将一个历史中的分支推送到一个新的远程仓库的 master 分支。

```
$ git branch history c6e1e95
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

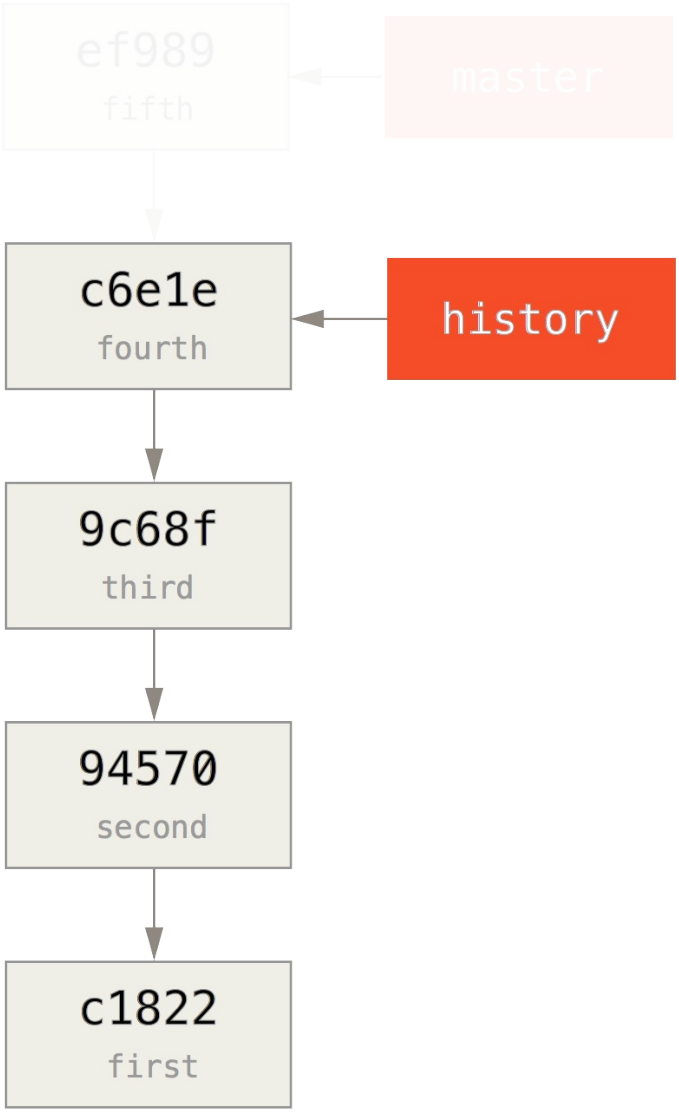


Figure 7-29.

现在我们可以把这个新的 `history` 分支推送到我们新仓库的 `master` 分支：

```
$ git remote add project-history https://github.com/schacon/project-history
$ git push project-history history:master
Counting objects: 12, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (12/12), 907 bytes, done.
Total 12 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (12/12), done.
To git@github.com:schacon/project-history.git
```

```
 * [new branch]      history -> master
```

这样一来，我们的历史版本就发布了。 稍难的部分则是删减我们最近的历史来让它变得更小。 我们需要一个重叠以便于用一个相等的提交来替换另一个提交，这样一来，我们将截断到第四、五个提交。

```
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

在这种情况下，创建一个能够指导扩展历史的基础提交是很有用的。 这样一来，如果其他的开发者想要修改第一次提交或者其他操作时就知道要做些什么，因此，接下来我们要做的是用命令创建一个最初的提交对象，然后将剩下的提交（第四、第五个提交）变基到它的上面。

为了这么做，我们需要选择一个点去拆分，对于我们而言是第三个提交（SHA 是 `9c68fdc`）。因此我们的提交将基于此提交树。我们可以使用 `commit-tree` 命令来创建基础提交，这样我们就有了一个树，并返回一个全新的、无父节点的 SHA 提交对象。

```
$ echo 'get history from blah blah blah' | git commit-tree 9c68fdc^{tree}
622e88e9cbfbacfb75b5279245b9fb38dfea10cf
```

> NOTE
>
> `commit-tree` 命令属于底层指令。有许多指令并非直接使用，而是被 其他的 Git 命令用来做更小一些的工作。有时当我们做一些像这样的奇怪事情时，它们允许我们做一些不适用于日常使用但真正底层的东西。更多关于底层命令的内容请参见 底层命令和高层命令

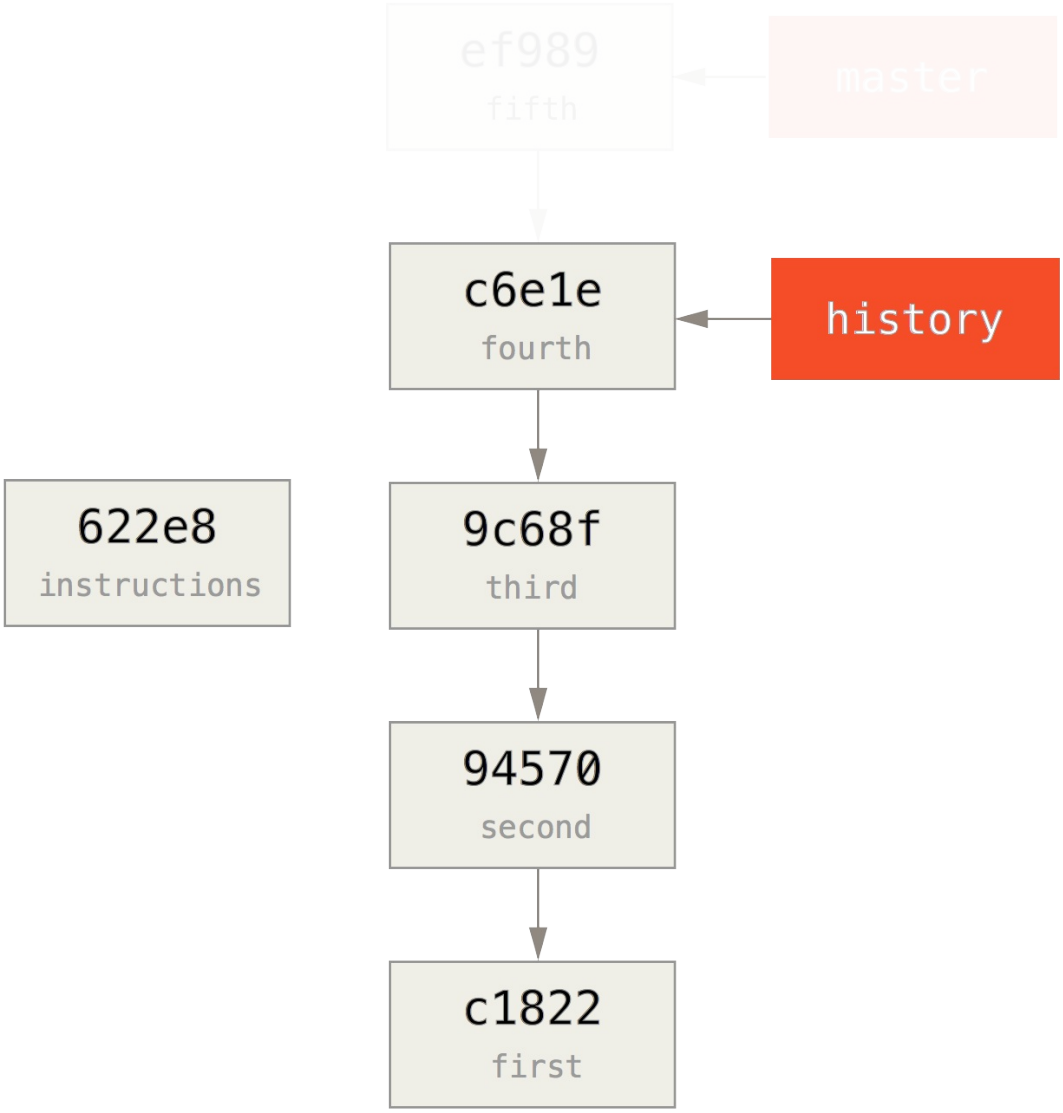


Figure 7-30.

现在我们已经有一个基础提交了，我们可以通过 `git rebase --onto` 命令来将剩余的历史变基到基础提交之上。 `--onto` 参数是刚才 `commit-tree` 命令返回的 SHA 值，变基点会成为第三个提交（我们想留下的第一个提交的父提交， `9c68fdc` ）：

```
$ git rebase --onto 622e88 9c68fdc
First, rewinding head to replay your work on top of it...
Applying: fourth commit
Applying: fifth commit
```

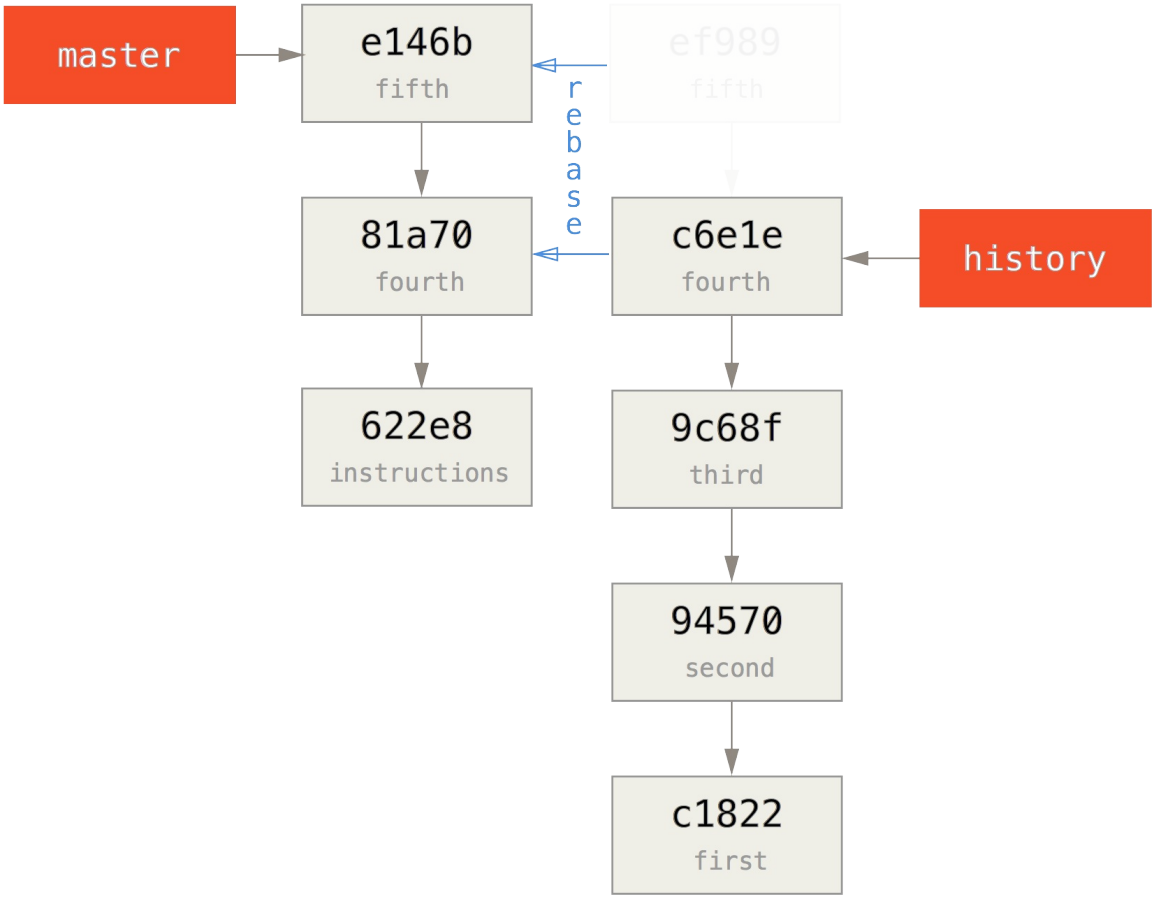


Figure 7-31.

我们已经用基础提交重写了最近的历史，基础提交包括如何重新组成整个历史的说明。 我们可以将新历史推送到新项目中，当其他人克隆这个仓库时，他们仅能看到最近两次提交以及一个包含上述说明的基础提交。

现在我们将以想获得整个历史的人的身份来初次克隆这个项目。 在克隆这个截断后的仓库后为了得到历史数据，需要添加第二个远程的历史版本库并对其做获取操作：

```
$ git clone https://github.com/schacon/project
$ cd project

$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git remote add project-history https://github.com/schacon/project-history
$ git fetch project-history
```

```
From https://github.com/schacon/project-history
 * [new branch]      master     -> project-history/master
```

现在，协作者在 `master` 分支中拥有他们最近的提交并且在 `project-history/master` 分支中拥有过去的提交。

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git log --oneline project-history/master
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

为了合并它们，你可以使用 `git replace` 命令加上你想替换的提交信息来进行替换。 这样一来，我们就可以将 master 分支中的第四个提交替换为 `project-history/master` 分支中的“第四个”提交。

```
$ git replace 81a708d c6e1e95
```

现在，查看 `master` 分支中的历史信息，显示如下：

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

很酷，是不是？不用改变上游的 SHA-1 我们就能用一个提交来替换历史中的所有不同的提交，并且所有的工具（ `bisect` ， `blame` 等）也都奏效。

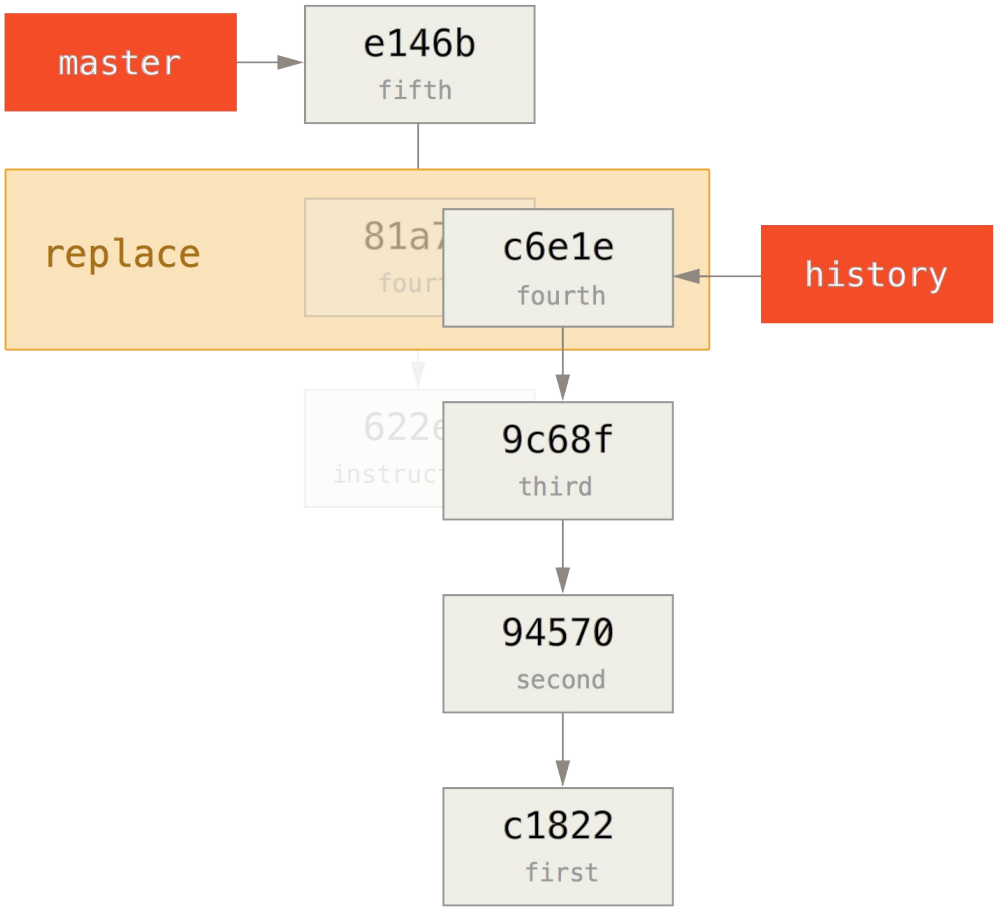


Figure 7-32.

有趣的是，即使是使用了 `c6e1e95` 提交数据来进行替换，它的 SHA-1 仍显示为 `81a708d`。 即使你运行了 `cat-file` 命令，它仍会显示你替换的数据：

```
$ git cat-file -p 81a708d
tree 7bc544cf438903b65ca9104a1e30345eee6c083d
parent 9c68fdceee073230f19ebb8b5e7fc71b479c0252
author Scott Chacon <schacon@gmail.com> 1268712581 -0700
committer Scott Chacon <schacon@gmail.com> 1268712581 -0700

fourth commit
```

请记住，`81a708d` 真正的父提交是 `622e882` 占位提交，而非呈现的 `9c68fdce` 提交。

另一个有趣的事情是数据将会以以下引用显示：

```
$ git for-each-ref
```

```
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/heads/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit refs/remotes/history/master
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/HEAD
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit refs/replace/81a708dd0e167a3f691541c7a6463343bc457040
```

这意味着我们可以轻而易举的和其他人分享替换，因为我们可以将替换推送到服务器中并且其他人可以轻松地下载。 也许在历史移植情况下不是很有用（既然每个人都乐意下载最新版本和历史版本，为何还要拆分他们呢？），但在其他情况下仍然很有用。

7.14 凭证存储

凭证存储

如果你使用的是 SSH 方式连接远端，并且设置了一个没有口令的密钥，这样就可以在不输入用户名和密码的情况下安全地传输数据。 然而，这对 HTTP 协议来说是不可能的 —— 每一个连接都是需要用户名和密码的。 这在使用双重认证的情况下会更麻烦，因为你需要输入一个随机生成并且毫无规律的 token 作为密码。

幸运的是，Git 拥有一个凭证系统来处理这个事情。 下面有一些 Git 的选项：

- 默认所有都不缓存。 每一次连接都会询问你的用户名和密码。
- “cache” 模式会将凭证存放在内存中一段时间。 密码永远不会被存储在磁盘中，并且在15分钟后从内存中清除。
- “store” 模式会将凭证用明文的形式存放在磁盘中，并且永不过期。 这意味着除非你修改了你在 Git 服务器上的密码，否则你永远不需要再次输入你的凭证信息。 这种方式的缺点是你的密码是用明文的方式存放在你的 home 目录下。
- 如果你使用的是 Mac，Git 还有一种 “osxkeychain” 模式，它会将凭证缓存到你系统用户的钥匙串中。 这种方式将凭证存放在磁盘中，并且永不过期，但是是被加密的，这种加密方式与存放 HTTPS 凭证以及 Safari 的自动填写是相同的。
- 如果你使用的是 Windows，你可以安装一个叫做 “winstore” 的辅助工具。 这和上面说的 “osxkeychain” 十分类似，但是是使用 Windows Credential Store 来控制敏感信息。 可以在 https://gitcredentialstore.codeplex.com 下载。

你可以设置 Git 的配置来选择上述的一种方式

```
$ git config --global credential.helper cache
```

部分辅助工具有一些选项。 “store” 模式可以接受一个 `--file <path>` 参数，可以自定义存放密码的文件路径（默认是 `~/.git-credentials`）。 “cache” 模式有 `--timeout <seconds>` 参数，可以设置后台进程的存活时间（默认是 “900”，也就是 15 分钟）。 下面是一个配置 “store” 模式自定义路径的例子：

```
$ git config --global credential.helper store --file ~/.my-credentials
```

Git 甚至允许你配置多个辅助工具。 当查找特定服务器的凭证时，Git 会按顺序查询，并且在找到第一个

回答时停止查询。 当保存凭证时，Git 会将用户名和密码发送给 所有 配置列表中的辅助工具，它们会按自己的方式处理用户名和密码。 如果你在闪存上有一个凭证文件，但又希望在该闪存被拔出的情况下使用内存缓存来保存用户名密码，`.gitconfig` 配置文件如下：

```
[credential]
    helper = store --file /mnt/thumbdrive/.git-credentials
    helper = cache --timeout 30000
```

底层实现

这些是如何实现的呢？ Git 凭证辅助工具系统的命令是 `git credential`，这个命令接收一个参数，并通过标准输入获取更多的参数。

举一个例子更容易理解。 我们假设已经配置好一个凭证辅助工具，这个辅助工具保存了 `mygithost` 的凭证信息。 下面是一个使用 “fill” 命令的会话，当 Git 尝试寻找一个服务器的凭证时就会被调用。

```
$ git credential fill
protocol=https
host=mygithost

protocol=https
host=mygithost
username=bob
password=s3cre7
$ git credential fill
protocol=https
host=unknownhost

Username for 'https://unknownhost': bob
Password for 'https://bob@unknownhost':
protocol=https
host=unknownhost
username=bob
password=s3cre7
```

❶这是开始交互的命令。

❷Git-credential 接下来会等待标准输入。 我们提供我们所知道的信息：协议和主机名。

❸一个空行代表输入已经完成，凭证系统应该输出它所知道的信息。

❹接下来由 Git-credential 接管，并且将找到的信息打印到标准输出。

❺如果没有找到对应的凭证，Git 会询问用户的用户名和密码，我们将这些信息输入到在标准输出的地方（这个例子中是同一个控制台）。

凭证系统实际调用的程序和 Git 本身是分开的；具体是哪一个以及如何调用与 `credential.helper` 配置的值有关。 这个配置有多种格式：

配置值	行为
`foo`	执行 `git-credential-foo`
`foo -a --opt=bcd`	执行 `git-credential-foo -a --opt=bcd`
`/absolute/path/foo -xyz`	执行 `/absolute/path/foo -xyz`
`!f() { echo "password=s3cre7"; }; f`	`!` 后面的代码会在shell执行

上面描述的辅助工具可以被称做 `git-credential-cache` 、 `git-credential-store` 之类，我们可以配置它们来接受命令行参数。 通常的格式是 “git-credential-foo [args] ” 标准输入/输出协议和 git-credential 一样，但它们使用的是一套稍微不太一样的行为：

- `get` 是请求输入一对用户名和密码。
- `store` 是请求保存一个凭证到辅助工具的内存。
- `erase` 会将给定的证书从辅助工具内存中清除。

对于 `store` 和 `erase` 两个行为是不需要返回数据的（Git 也会忽略掉）。 然而对于 `get` ，Git 对辅助工具的返回信息十分感兴趣。

如果辅助工具没有任何有用的信息，它可以直接退出而不需要输出任何东西，但如果它有这些信息，它在提供的信息后面增加它所拥有的信息。 这些输出会被视为一系列的赋值语句；每一个提供的数据都会将 Git 已有的数据替换掉。

这有一个和上面一样的例子，但是跳过了 git-credential 这一步，直接到 git-credential-store:

```
$ git credential-store --file ~/git.store store
protocol=https
host=mygithost
username=bob
password=s3cre7
$ git credential-store --file ~/git.store get
protocol=https
host=mygithost

username=bob
password=s3cre7
```

❶我们告诉 `git-credential-store` 去保存凭证：当访问 `https://mygithost` 时使用用户名 “bob” ，密码是 “s3cre7” 。

❷现在我们取出这个凭证。 我们提供连接这部分的信息（ `https://mygithost` ）以及一个空行。

❸ `git-credential-store` 输出我们之前保存的用户名和密码。

`~/git.store` 文件的内容类似：

```
https://bob:s3cre7@mygithost
```

仅仅是一系列包含凭证信息URL组成的行。 `osxkeychain` 和 `winstore` 辅助工具使用它们后端存储的原生格式，而 `cache` 使用它的内存格式（其他进程无法读取）。

自定义凭证缓存

已经知道 `git-credential-store` 之类的是和 Git 是相互独立的程序，就不难理解 Git 凭证辅助工具可以是 任意 程序。 虽然 Git 提供的辅助工具覆盖了大多数常见的使用场景，但并不能满足所有情况。 比如，假设你的整个团队共享一些凭证，也许是在部署时使用。 这些凭证是保存在一个共享目录里，由于这些凭证经常变更，所以你不想把它们复制到你自己的凭证仓库中。 现有的辅助工具无法满足这种情况；来看看我们如何自己实现一个。 这个程序应该拥有几个核心功能：

1. 我们唯一需要关注的行为是 `get`；`store` 和 `erase` 是写操作，所以当接受到这两个请求时我们直接退出即可。
2. 共享的凭证文件格式和 `git-credential-store` 使用的格式相同。
3. 凭证文件的路径一般是固定的，但我们应该允许用户传入一个自定义路径以防万一。

我们再一次使用 Ruby 来编写这个扩展，但只要 Git 能够执行最终的程序，任何语言都是可以的。 这是我们的凭证辅助工具的完整代码：

```
#!/usr/bin/env ruby

require 'optparse'

path = File.expand_path '~/.git-credentials'
OptionParser.new do |opts|
    opts.banner = 'USAGE: git-credential-read-only [options] <action>'
    opts.on('-f', '--file PATH', 'Specify path for backing store') do |argpath|
        path = File.expand_path argpath
    end
end.parse!

exit(0) unless ARGV[0].downcase == 'get'
exit(0) unless File.exists? path

known = {}
while line = STDIN.gets
    break if line.strip == ''
    k,v = line.strip.split '=', 2
    known[k] = v
end

File.readlines(path).each do |fileline|
    prot,user,pass,host = fileline.scan(/^(.*?):\/\/(.*?):(.*?)@(.*)$/).first
    if prot == known['protocol'] and host == known['host'] then
        puts "protocol=#{prot}"
        puts "host=#{host}"
        puts "username=#{user}"
        puts "password=#{pass}"
```

```
        exit(0)
    end
end
```

❶我们在这里解析命令行参数，允许用户指定输入文件，默认是 `~/.git-credentials`.

❷这个程序只有在接受到 `get` 行为的请求并且后端存储的文件存在时才会有输出。

❸这个循环从标准输入读取数据，直到读取到第一个空行。 输入的数据被保存到 `known` 哈希表中，之后需要用到。

❹这个循环读取存储文件中的内容，寻找匹配的行。 如果 `known` 中的协议和主机名与该行相匹配，这个程序输出结果并退出。

我们把这个辅助工具保存为 `git-credential-read-only`，放到我们的 `PATH` 路径下并且给予执行权限。 一个交互式会话类似：

```
$ git credential-read-only --file=/mnt/shared/creds get
protocol=https
host=mygithost

protocol=https
host=mygithost
username=bob
password=s3cre7
```

由于这个的名字是 “git-” 开头，所以我们可以在配置值中使用简便的语法：

```
$ git config --global credential.helper read-only --file /mnt/shared/creds
```

正如你看到的，扩展这个系统是相当简单的，并且可以为你和你的团队解决一些常见问题。

7.15 总结

总结

你已经接触了很多能够精确地操控提交和暂存区的高级工具。 当你碰到问题时，你应该可以很容易找出是哪个分支在什么时候由谁引入了它们。 如果你想在项目中使用子项目，你也已经知道如何来满足这些需求。 到此，你应该能毫无压力地在命令行中使用 Git 来完成日常中的大部分事情。

8. 自定义 Git

第8章. 自定义 Git

到目前为止，我们已经阐述了 Git 基本的运作机制和使用方式，介绍了许多 Git 提供的工具来帮助你简单且有效地使用它。在本章，我们将演示如何借助 Git 的一些重要的配置方法和钩子机制，来满足自定义的需求。通过这些工具，它会和你、你的公司或你的团队配合得天衣无缝。

8.1 配置 Git

配置 Git

你在 起步 中看到，可以用 `git config` 配置 Git。 首先要做的事情就是设置你的名字和邮件地址：

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

现在，你会了解到许多更有趣的选项，并用类似的方式来定制 Git。

首先，快速回忆下：Git 使用一系列配置文件来保存你自定义的行为。 它首先会查找 `/etc/gitconfig` 文件，该文件含有系统里每位用户及他们所拥有的仓库的配置值。 如果你传递 `--system` 选项给 `git config`，它就会读写该文件。

接下来 Git 会查找每个用户的 `~/.gitconfig` 文件（或者 `~/.config/git/config` 文件）。 你可以传递 `--global` 选项让 Git 读写该文件。

最后 Git 会查找你正在操作的版本库所对应的 Git 目录下的配置文件（ `.git/config` ）。 这个文件中的值只对该版本库有效。

以上三个层次中每层的配置（系统、全局、本地）都会覆盖掉上一层次的配置，所以 `.git/config` 中的值会覆盖掉 `/etc/gitconfig` 中所对应的值。

> NOTE
>
> Git 的配置文件是纯文本的，所以你可以直接手动编辑这些配置文件，输入合乎语法的值。 但是运行 `git config` 命令会更简单些。

客户端基本配置

Git 能够识别的配置项分为两大类：客户端和服务器端。 其中大部分属于客户端配置 —— 可以依你个人的工作偏好进行配置。 尽管 Git 支持的选项 繁多，但其中大部分仅仅在某些罕见的情况下有意义。 我们只讲述最平常和最有用的选项。 如果想得到你当前版本的 Git 支持的选项列表，请运行

```
$ man git-config
```

这个命令列出了所有可用的选项，以及与之相关的介绍。 你也可以在 http://git-scm.com/docs/git-config.html 找到同样的内容。

`core.editor`

默认情况下，Git 会调用环境变量（ `$VISUAL` 或 `$EDITOR` ）设置的任意文本编辑器，如果没有设置，会调用 `vi` 来创建和编辑你的提交以及标签信息。 你可以使用 `core.editor` 选项来修改默认的编辑器：

```
$ git config --global core.editor emacs
```

现在，无论你定义了什么终端编辑器，Git 都会调用 Emacs 编辑信息。

`commit.template`
如果把此项指定为你的系统上某个文件的路径，当你提交的时候， Git 会使用该文件的内容作为提交的默认信息。 例如：假设你创建了一个叫 `~/.gitmessage.txt` 的模板文件，类似这样：

```
subject line

what happened

[ticket: X]
```

要想让 Git 把它作为运行 `git commit` 时显示在你的编辑器中的默认信息， 如下设置 `commit.template` ：

```
$ git config --global commit.template ~/.gitmessage.txt
$ git commit
```

然后当你提交时，编辑器中就会显示如下的提交信息占位符：

```
subject line

what happened

[ticket: X]
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   lib/test.rb
#
~
~
".git/COMMIT_EDITMSG" 14L, 297C
```

如果你的团队对提交信息有格式要求，可以在系统上创建一个文件，并配置 Git 把它作为默认的模板，这样可以更加容易地使提交信息遵循格式。

`core.pager`

该配置项指定 Git 运行诸如 `log` 和 `diff` 等命令所使用的分页器。 你可以把它设置成用 `more` 或者任何你喜欢的分页器（默认用的是 `less` ），当然也可以设置成空字符串，关闭该选项：

```
$ git config --global core.pager ''
```

这样不管命令的输出量多少，Git 都会在一页显示所有内容。

`user.signingkey`
如果你要创建经签署的含附注的标签（正如 签署工作 所述），那么把你的 GPG 签署密钥设置为配置项会更好。 如下设置你的密钥 ID：

```
$ git config --global user.signingkey <gpg-key-id>
```

现在，你每次运行 `git tag` 命令时，即可直接签署标签，而无需定义密钥：

```
$ git tag -s <tag-name>
```

`core.excludesfile`
正如 忽略文件 所述，你可以在你的项目的 `.gitignore` 文件里面规定无需纳入 Git 管理的文件的模板，这样它们既不会出现在未跟踪列表，也不会在你运行 `git add` 后被暂存。

不过有些时候，你想要在你所有的版本库中忽略掉某一类文件。 如果你的操作系统是 OS X，很可能就是指 `.DS_Store` 。 如果你把 Emacs 或 Vim 作为首选的编辑器，你肯定知道以 `~` 结尾的临时文件。

这个配置允许你设置类似于全局生效的 `.gitignore` 文件。 如果你按照下面的内容创建一个 `~/.gitignore_global` 文件：

```
*~
.DS_Store
```

……然后运行 `git config --global core.excludesfile ~/.gitignore_global` ，Git 将把那些文件永远地拒之门外。

`help.autocorrect`
假如你打错了一条命令，会显示：

```
$ git chekcout master
git：'chekcout' 不是一个 git 命令。参见 'git --help'。

您指的是这个么？
  checkout
```

Git 会尝试猜测你的意图，但是它不会越俎代庖。 如果你把 `help.autocorrect` 设置成 1，那么只要有一个命令被模糊匹配到了，Git 会自动运行该命令。

```
$ git chekcout master
警告：您运行一个不存在的 Git 命令 'chekcout'。继续执行假定您要要运行的
是 'checkout'
在 0.1 秒钟后自动运行...
```

注意提示信息中的“0.1 秒”。 `help.autocorrect` 接受一个代表十分之一秒的整数。 所以如果你把它设置为 50, Git 将在自动执行命令前给你 5 秒的时间改变主意。

Git 中的着色

Git 充分支持对终端内容着色，对你凭肉眼简单、快速分析命令输出有很大帮助。 你可以设置许多的相关选项来满足自己的偏好。

`color.ui`

Git 会自动着色大部分输出内容，但如果你不喜欢花花绿绿，也可以关掉。 要想关掉 Git 的终端颜色输出，试一下这个：

```
$ git config --global color.ui false
```

这个设置的默认值是 `auto`，它会着色直接输出到终端的内容；而当内容被重定向到一个管道或文件时，则忽略着色功能。

你也可以设置成 `always`，来忽略掉管道和终端的不同，即在任何情况下着色输出。 你很少会这么设置，在大多数场合下，如果你想在被重定向的输出中插入颜色码，可以传递 `--color` 标志给 Git 命令来强制它这么做。 默认设置就已经能满足大多数情况下的需求了。

`color.*`

要想具体到哪些命令输出需要被着色以及怎样着色，你需要用到和具体命令有关的颜色配置选项。 它们都能被置为 `true`、 `false` 或 `always`：

```
color.branch
color.diff
color.interactive
color.status
```

另外，以上每个配置项都有子选项，它们可以被用来覆盖其父设置，以达到为输出的各个部分着色的目的。 例如，为了让 `diff` 的输出信息以蓝色前景、黑色背景和粗体显示，你可以运行

```
$ git config --global color.diff.meta "blue black bold"
```

你能设置的颜色有：`normal`、`black`、`red`、`green`、`yellow`、`blue`、`magenta`、`cyan` 或 `white`。 正如以上例子设置的粗体属性，想要设置字体属性的话，可以选择包括：`bold`、`dim`、`ul`（下划线）、`blink`、`reverse`（交换前景色和背景色）。

外部的合并与比较工具

虽然 Git 自己内置了一个 `diff` 实现，而且到目前为止我们一直在使用它，但你能够用一个外部的工具替代它。 除此以外，你还能设置一个图形化的工具来合并和解决冲突，从而不必自己手动解决。 这里我们以一个不错且免费的工具 —— Perforce 图形化合并工具（P4Merge） —— 来展示如何用一个外部的工具来合并和解决冲突。

P4Merge 可以在所有主流平台上运行，所以安装上应该没有什么困难。 在这个例子中，我们使用的路径名可以直接应用在 Mac 和 Linux 上；在 Windows 上，`/usr/local/bin` 需要被改为你的环境中可执行文件所在的目录路径。

首先， 从 http://www.perforce.com/downloads/Perforce/ 下载 P4Merge。 接下来，你要编写一个全局包装脚本来运行你的命令。 我们会使用 Mac 上的路径来指定该脚本的位置，在其他系统上，它将是 `p4merge` 二进制文件所在的目录。 创建一个名为 `extMerge` 的脚本包装 merge 命令，让它把参数转发给 p4merge 二进制文件：

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

包装 diff 命令的脚本首先确保传递了七个参数过来，随后把其中两个转发给包装了 merge 的脚本。 默认情况下， Git 传递以下参数给 `diff`：

```
path old-file old-hex old-mode new-file new-hex new-mode
```

由于你仅仅需要 `old-file` 和 `new-file` 参数，由包装 diff 的脚本来转发它们吧。

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

你也需要确保这些脚本具有可执行权限：

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

现在你可以修改配置文件来使用你自定义的合并和比较工具了。 这将涉及许多自定义设置：`merge.tool` 通知 Git 该使用哪个合并工具， `mergetool.<tool>.cmd` 规定命令运行的方式，

`mergetool.<tool>.trustExitCode` 会通知 Git 程序的返回值是否表示合并操作成功，`diff.external` 通知 Git 该用什么命令做比较。 因此，你可以运行以下四条配置命令：

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
  'extMerge \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"'
$ git config --global mergetool.extMerge.trustExitCode false
$ git config --global diff.external extDiff
```

或编辑你的 `~/.gitconfig` 文件，添加以下各行：

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
  trustExitCode = false
[diff]
  external = extDiff
```

待一切设置妥当后，如果你像这样运行 diff 命令：

```
$ git diff 32d1776b1^ 32d1776b1
```

Git 将启动 P4Merge，而不是在命令行输出比较的结果，就像这样：

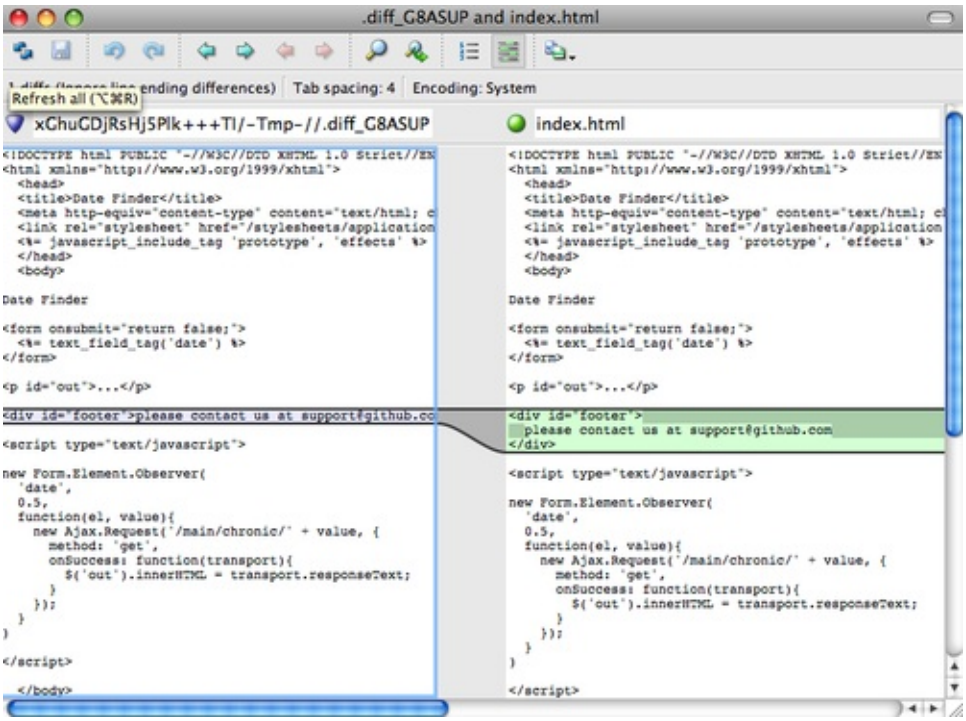


Figure 8-1. P4Merge.

如果你尝试合并两个分支，随后遇到了合并冲突，运行 `git mergetool`，Git 会调用 P4Merge 让你通过图形界面来解决冲突。

设置包装脚本的好处在于大大降低了改变 diff 和 merge 工具的工作量。 举个例子，想把 `extDiff` 和 `extMerge` 的工具改成 KDiff3，你要做的仅仅是编辑 `extMerge` 脚本文件：

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

现在，Git 将使用 KDiff3 作为查看比较和解决合并冲突的工具。

Git 预设了许多其他的合并和解决冲突的工具，无需特别的设置你就能用上它们。 要想看到它支持的工具列表，试一下这个：

```
$ git mergetool --tool-help
'git mergetool --tool=<tool>' may be set to one of the following:
        emerge
        gvimdiff
        gvimdiff2
        opendiff
        p4merge
        vimdiff
        vimdiff2

The following tools are valid, but not currently available:
        araxis
        bc3
        codecompare
        deltawalker
        diffmerge
        diffuse
        ecmerge
        kdiff3
        meld
        tkdiff
        tortoisemerge
        xxdiff

Some of the tools listed above only work in a windowed
environment. If run in a terminal-only session, they will fail.
```

如果你不想用到 KDiff3 的所有功能，只是想用它来合并，那么 kdiff3 正符合你的要求，运行：

```
$ git config --global merge.tool kdiff3
```

如果运行了以上命令，而没有设置 `extMerge` 和 `extDiff` 文件，Git 会用 KDiff3 做合并，让内置的 `diff` 来做比较。

格式化与多余的空白字符

格式化与多余的空白字符是许多开发人员在协作时，特别是在跨平台情况下，不时会遇到的令人头疼的琐碎的问题。 由于编辑器的不同或者文件行尾的换行符在 Windows 下被替换了，一些细微的空格变化会不经意地混入提交的补丁或其它协作成果中。 不用怕，Git 提供了一些配置项来帮助你解决这些问题。

`core.autocrlf`

假如你正在 Windows 上写程序，而你的同伴用的是其他系统（或相反），你可能会遇到 CRLF 问题。 这是因为 Windows 使用回车（CR）和换行（LF）两个字符来结束一行，而 Mac 和 Linux 只使用换行（LF）一个字符。 虽然这是小问题，但它会极大地扰乱跨平台协作。许多 Windows 上的编辑器会悄悄把行尾的换行字符转换成回车和换行，或在用户按下 Enter 键时，插入回车和换行两个字符。

Git 可以在你提交时自动地把回车和换行转换成换行，而在检出代码时把换行转换成回车和换行。 你可以用 `core.autocrlf` 来打开此项功能。 如果是在 Windows 系统上，把它设置成 `true`，这样在检出代码时，换行会被转换成回车和换行：

```
$ git config --global core.autocrlf true
```

如果使用以换行作为行结束符的 Linux 或 Mac，你不需要 Git 在检出文件时进行自动的转换；然而当一个以回车加换行作为行结束符的文件不小心被引入时，你肯定想让 Git 修正。 你可以把 `core.autocrlf` 设置成 input 来告诉 Git 在提交时把回车和换行转换成换行，检出时不转换：

```
$ git config --global core.autocrlf input
```

这样在 Windows 上的检出文件中会保留回车和换行，而在 Mac 和 Linux 上，以及版本库中会保留换行。

如果你是 Windows 程序员，且正在开发仅运行在 Windows 上的项目，可以设置 `false` 取消此功能，把回车保留在版本库中：

```
$ git config --global core.autocrlf false
```

`core.whitespace`

Git 预先设置了一些选项来探测和修正多余空白字符问题。 它提供了六种处理多余空白字符的主要选项 —— 其中三个默认开启，另外三个默认关闭，不过你可以自由地设置它们。

默认被打开的三个选项是：`blank-at-eol`，查找行尾的空格；`blank-at-eof`，盯住文件底部的空行；`space-before-tab`，警惕行头 tab 前面的空格。

默认被关闭的三个选项是：`indent-with-non-tab`，揪出以空格而非 tab 开头的行（你可以用 `tabwidth` 选项控制它）；`tab-in-indent`，监视在行头表示缩进的 tab；`cr-at-eol`，告诉 Git 忽略行尾的回车。

通过设置 `core.whitespace`，你可以让 Git 按照你的意图来打开或关闭以逗号分割的选项。 要想关闭某个选项，你可以在输入设置选项时不指定它或在它前面加个 `-`。 例如，如果你想要打开除 `cr-at-eol` 之外的所有选项：

```
$ git config --global core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab
```

当你运行 `git diff` 命令并尝试给输出着色时，Git 将探测到这些问题，因此你在提交前就能修复它们。 用 `git apply` 打补丁时你也会从中受益。 如果正准备应用的补丁存有特定的空白问题，你可以让 Git 在应用补丁时发出警告：

```
$ git apply --whitespace=warn <patch>
```

或者让 Git 在打上补丁前自动修正此问题：

```
$ git apply --whitespace=fix <patch>
```

这些选项也能运用于 `git rebase`。 如果提交了有空白问题的文件，但还没推送到上游，你可以运行 `git rebase --whitespace=fix` 来让 Git 在重写补丁时自动修正它们。

服务器端配置

Git 服务器端的配置项相对来说并不多，但仍有一些饶有生趣的选项值得你一看。

`receive.fsckObjects`

Git 能够确认每个对象的有效性以及 SHA-1 检验和是否保持一致。 但 Git 不会在每次推送时都这么做。 这个操作很耗时间，很有可能会拖慢提交的过程，特别是当库或推送的文件很大的情况下。 如果想在每次推送时都要求 Git 检查一致性，设置 `receive.fsckObjects` 为 true 来强迫它这么做：

```
$ git config --system receive.fsckObjects true
```

现在 Git 会在每次推送生效前检查库的完整性，确保没有被有问题的客户端引入破坏性数据。

`receive.denyNonFastForwards`

如果你变基已经被推送的提交，继而再推送，又或者推送一个提交到远程分支，而这个远程分支当前指向的提交不在该提交的历史中，这样的推送会被拒绝。 这通常是个很好的策略，但有时在变基的过程中，你确信自己需要更新远程分支，可以在 push 命令后加 `-f` 标志来强制更新（force-update）。

要禁用这样的强制更新推送（force-pushes），可以设置 `receive.denyNonFastForwards`：

```
$ git config --system receive.denyNonFastForwards true
```

稍后我们会提到，用服务器端的接收钩子也能达到同样的目的。 那种方法可以做到更细致的控制，例如禁止某一类用户做非快进（non-fast-forwards）推送。

`receive.denyDeletes`

有一些方法可以绕过 `denyNonFastForwards` 策略。其中一种是先删除某个分支，再连同新的引用一起推送回该分支。 把 `receive.denyDeletes` 设置为 true 可以把这个漏洞补上：

```
$ git config --system receive.denyDeletes true
```

这样会禁止通过推送删除分支和标签 — 没有用户可以这么做。 要删除远程分支，必须从服务器手动删除引用文件。 通过用户访问控制列表（ACL）也能够在用户级的粒度上实现同样的功能，你将在 使用强制策略的一个例子 一节学到具体的做法。

8.2 Git 属性

Git 属性

你也可以针对特定的路径配置某些设置项，这样 Git 就只对特定的子目录或子文件集运用它们。 这些基于路径的设置项被称为 Git 属性，可以在你的目录下的 `.gitattributes` 文件内进行设置（通常是你的项目的根目录）。如果不想让这些属性文件与其它文件一同提交，你也可以在 `.git/info/attributes` 文件中进行设置。

通过使用属性，你可以对项目中的文件或目录单独定义不同的合并策略，让 Git 知道怎样比较非文本文件，或者让 Git 在提交或检出前过滤内容。 在本节，你将学习到一些能在自己的项目中用到的属性，并看到几个实际的例子。

二进制文件

你可以用 Git 属性让 Git 知道哪些是二进制文件（以防它没有识别出来），并指示其如何处理这些文件。 例如，一些文本文件是由机器产生的，没有办法进行比较，但是一些二进制文件可以比较。 你将了解到怎样让 Git 区分这些文件。

识别二进制文件

有些文件表面上是文本文件，实质上应被作为二进制文件处理。 例如，Mac 平台上的 Xcode 项目会包含一个以 `.pbxproj` 结尾的文件，它通常是一个记录项目构建配置等信息的 JSON（纯文本 Javascript 数据类型）数据集，由 IDE 写入磁盘。 虽然技术上看它是由 UTF-8 编码的文本文件，但你并不会希望将它当作文本文件来处理，因为它其实是一个轻量级数据库——如果有两个人修改了它，你通常无法合并内容，diff 的输出也帮不上什么忙。 它本应被机器处理。 因此，你想把它当成二进制文件。

要让 Git 把所有 `pbxproj` 文件当成二进制文件，在 `.gitattributes` 文件中如下设置：

```
*.pbxproj binary
```

现在，Git 不会尝试转换或修正回车换行（CRLF）问题，当你在项目中运行 `git show` 或 `git diff` 时，Git 也不会比较或打印该文件的变化。

比较二进制文件

你也可以使用 Git 属性来有效地比较两个二进制文件。 秘诀在于，告诉 Git 怎么把你的二进制文件转化为文本格式，从而能够使用普通的 diff 方式进行对比。

首先，让我们尝试用这个技术解决世人最头疼的问题之一：对 Microsoft Word 文档进行版本控制。 大家都知道，Microsoft Word 几乎是世上最难缠的编辑器，尽管如此，大家还是在用它。 如果想对 Word 文档进行版本控制，你可以把文件加入到 Git 库中，每次修改后提交即可。但这样做有什么实际意义呢？ 毕竟运行 `git diff` 命令后，你只能得到如下的结果：

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 88839c4..4afcb7c 100644
Binary files a/chapter1.docx and b/chapter1.docx differ
```

除了检出之后睁大眼睛逐行扫描，就真的没有办法直接比较两个不同版本的 Word 文档吗？ Git 属性能很好地解决此问题。 把下面这行文本加到你的 `.gitattributes` 文件中：

```
*.docx diff=word
```

这告诉 Git 当你尝试查看包含变更的比较结果时，所有匹配 `.docx` 模式的文件都应该使用“word”过滤器。 “word”过滤器是什么？ 我们现在就来设置它。 我们会对 Git 进行配置，令其能够借助 `docx2txt` 程序将 Word 文档转为可读文本文件，这样不同的文件间就能够正确比较了。

首先，你需要安装 `docx2txt` ；它可以从 http://docx2txt.sourceforge.net 下载。 按照 `INSTALL` 文件的说明，把它放到你的可执行路径下。 接下来，你还需要写一个脚本把输出结果包装成 Git 支持的格式。 在你的可执行路径下创建一个叫 `docx2txt` 文件，添加这些内容：

```
#!/bin/bash
docx2txt.pl $1 -
```

别忘了用 `chmod a+x` 给这个文件加上可执行权限。 最后，你需要配置 Git 来使用这个脚本：

```
$ git config diff.word.textconv docx2txt
```

现在如果在两个快照之间进行比较，Git 就会对那些以 `.docx` 结尾的文件应用“word”过滤器，即 `docx2txt` 。 这样你的 Word 文件就能被高效地转换成文本文件并进行比较了。

作为例子，我把本书的第一章另存为 Word 文件，并提交到 Git 版本库。 接着，往其中加入一个新的段落。 运行 `git diff` ，输出如下：

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 0b013ca..ba25db5 100644
--- a/chapter1.docx
+++ b/chapter1.docx
@@ -2,6 +2,7 @@
 This chapter will be about getting started with Git. We will begin at the beginning by explaining some
 background on version control tools, then move on to how to get Git running on your system and finally
 how to get it setup to start working with. At the end of this chapter you should understand why Git is
 around, why you should use it and you should be all setup to do so.
 1.1. About Version Control
 What is "version control", and why should you care? Version control is a system that records changes t
o a file or set of files over time so that you can recall specific versions later. For the examples in
 this book you will use software source code as the files being version controlled, though in reality yo
```

```
u can do this with nearly any type of file on a computer.
+Testing: 1, 2, 3.
 If you are a graphic or web designer and want to keep every version of an image or layout (which you w
ould most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you
 to revert files back to a previous state, revert the entire project back to a previous state, compare
changes over time, see who last modified something that might be causing a problem, who introduced an i
ssue and when, and more. Using a VCS also generally means that if you screw things up or lose files, yo
u can easily recover. In addition, you get all this for very little overhead.
 1.1.1. Local Version Control Systems
 Many people's version-control method of choice is to copy files into another directory (perhaps a time
-stamped directory, if they're clever). This approach is very common because it is so simple, but it is
 also incredibly error prone. It is easy to forget which directory you're in and accidentally write to
the wrong file or copy over files you don't mean to.
```

Git 成功地挑出了我们添加的那句话“Testing: 1, 2, 3.”，一字不差。 还算不上完美——格式上的变动显示不出来——但已经足够了。

你还能用这个方法比较图像文件。 其中一个办法是，在比较时对图像文件运用一个过滤器，提炼出 EXIF 信息——这是在大部分图像格式中都有记录的一种元数据。 如果你下载并安装了 `exiftool` 程序，可以利用它将图像转换为关于元数据的文本信息，这样比较时至少能以文本的形式显示发生过的变动：

```
$ echo '*.png diff=exif' >> .gitattributes
$ git config diff.exif.textconv exiftool
```

如果在项目中替换了一个图像文件，运行 `git diff` 命令的结果如下：

```
diff --git a/image.png b/image.png
index 88839c4..4afcb7c 100644
--- a/image.png
+++ b/image.png
@@ -1,12 +1,12 @@
 ExifTool Version Number         : 7.74
-File Size                       : 70 kB
-File Modification Date/Time     : 2009:04:21 07:02:45-07:00
+File Size                       : 94 kB
+File Modification Date/Time     : 2009:04:21 07:02:43-07:00
 File Type                       : PNG
 MIME Type                       : image/png
-Image Width                     : 1058
-Image Height                    : 889
+Image Width                     : 1056
+Image Height                    : 827
 Bit Depth                       : 8
 Color Type                      : RGB with Alpha
```

你一眼就能看出文件大小和图像尺寸发生了变化。

关键字展开

SVN 或 CVS 风格的关键字展开（keyword expansion）功能经常会被习惯于上述系统的开发者使用到。 在 Git 中，这项功能有一个主要问题，就是你无法利用它往文件中加入其关联提交的相关信息，因为 Git 总是先对文件做校验和运算（译者注：Git 中提交对象的校验依赖于文件的校验和，而 Git 属性针对特定

文件或路径，因此基于 Git 属性的关键字展开无法仅根据文件反推出对应的提交）。 不过，我们可以在检出某个文件后对其注入文本，并在再次提交前删除这些文本。 Git 属性提供了两种方法来达到这一目的。

一种方法是，你可以把文件所对应数据对象的 SHA-1 校验和自动注入到文件中的 `$Id$` 字段。 如果在一个或多个文件上设置了该属性，下次当你检出相关分支的时候，Git 会用相应数据对象的 SHA-1 值替换上述字段。 注意，这不是提交对象的 SHA-1 校验和，而是数据对象本身的校验和：

```
$ echo '*.txt ident' >> .gitattributes
$ echo '$Id$' > test.txt
```

当你下次检出文件时，Git 将注入数据对象的 SHA-1 校验和：

```
$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

然而，这个结果的用途比较有限。 如果用过 CVS 或 Subversion 的关键字替换功能，我们会想加上一个时间戳信息——光有 SHA-1 校验和用途不大，它仅仅是个随机字符串，你无法凭字面值来区分不同 SHA-1 时间上的先后。

因此 Git 属性提供了另一种方法：我们可以编写自己的过滤器来实现文件提交或检出时的关键字替换。 一个过滤器由“clean”和“smudge”两个子过滤器组成。 在 `.gitattributes` 文件中，你能对特定的路径设置一个过滤器，然后设置文件检出前的处理脚本（“smudge”，见 Figure 8-2）和文件暂存前的处理脚本（“clean”，见 Figure 8-3）。 这两个过滤器能够被用来做各种有趣的事。

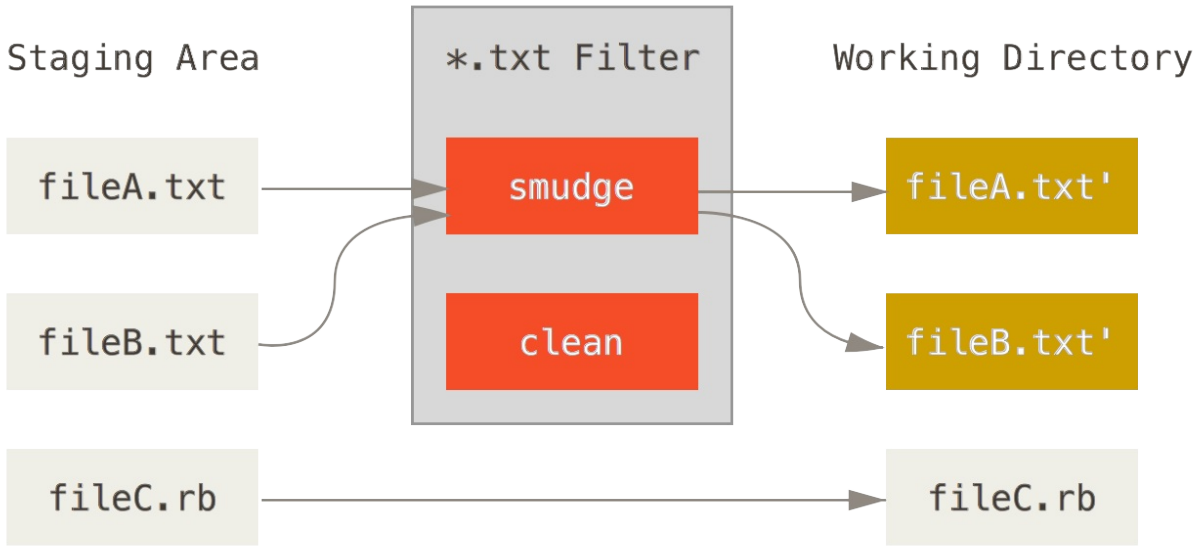


Figure 8-2. “smudge”过滤器会在文件被检出时触发

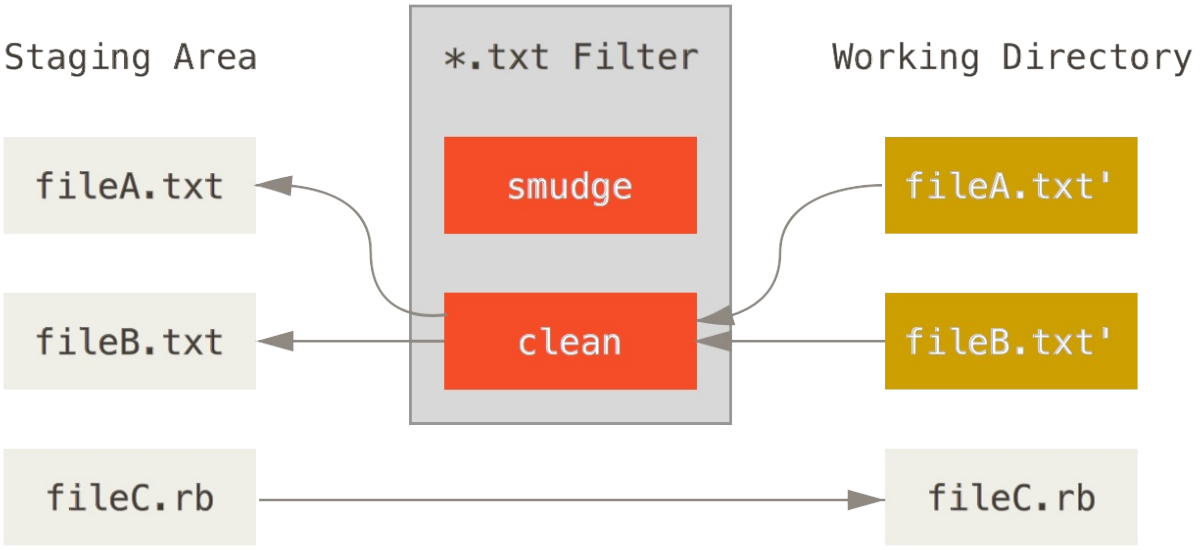


Figure 8-3. “clean”过滤器会在文件被暂存时触发

在（Git 源码中）实现这个特性的原始提交信息里给出了一个简单的例子：在提交前，用 `indent` 程序过滤所有 C 源码。 你可以在 `.gitattributes` 文件中对 filter 属性设置“indent”过滤器来过滤 `*.c` 文件

```
*.c filter=indent
```

然后，通过以下配置，让 Git 知道“indent”过滤器在 smudge 和 clean 时分别该做什么：

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

在这个例子中，当你暂存 `*.c` 文件时， `indent` 程序会先被触发；在把它们检出回硬盘时， `cat` 程序会先被触发。 `cat` 在这里没什么实际作用：它仅仅把输入的数据重新输出。 这样的组合可以有效地在暂存前用 `indent` 过滤所有的 C 源码。

另一个有趣的例子是实现 RCS 风格的 `$Date$` 关键字展开。 要想演示这个例子，我们需要实现这样的一个小脚本：接受文件名参数，得到项目的最新提交日期，并把日期写入该文件。 下面是一个实现了该功能的 Ruby 小脚本：

```
#! /usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

这个脚本从 `git log` 中得到最新提交日期，将其注入所有输入文件的 `$Date$` 字段，并输出结果——你可以使用最顺手的语言轻松实现一个类似的脚本。 把该脚本命名为 `expand_date`，放到你的可执行路径中。 现在，你需要在 Git 中设置一个过滤器（就叫它 `dater` 吧），让它在检出文件时调用你的 `expand_date` 来注入时间戳，完成 smudge 操作。 暂存文件时的 clean 操作则是用一行 Perl 表达式清除注入的内容：

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\\\$Date[^\\\$]*\\\$/\\\$Date\\\$/"'
```

这段 Perl 代码会删除 `$Date$` 后面注入的内容，恢复它的原貌。 过滤器终于准备完成了，是时候测试一下。创建一个带有 `$Date$` 关键字的文件，然后给它设置一个 Git 属性，关联我们的新过滤器：

```
$ echo '# $Date$' > date_test.txt
$ echo 'date*.txt filter=dater' >> .gitattributes
```

提交该文件，并再次检出，你会发现关键字如期被替换了：

```
$ git add date_test.txt .gitattributes
$ git commit -m "Testing date expansion in Git"
$ rm date_test.txt
$ git checkout date_test.txt
$ cat date_test.txt
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

自定义过滤器真的很强大。 不过你需要注意的是，因为 `.gitattributes` 文件会随着项目一起提交，而过滤器（例如这里的 `dater`）不会，所以过滤器有可能会失效。 当你在设计这些过滤器时，要注重容错性——它们在出错时应该能优雅地退出，从而不至于影响项目的正常运行。

导出版本库

Git 属性在导出项目归档（archive）时也能发挥作用。

`export-ignore`

当归档的时候，可以设置 Git 不导出某些文件和目录。 如果你不想在归档中包含某个子目录或文件，但想把它们纳入项目的版本管理中，你可以在 `export-ignore` 属性中指定它们。

例如，假设你在 `test/` 子目录下有一些测试文件，不希望它们被包含在项目导出的压缩包（tarball）中。 你可以增加下面这行到 Git 属性文件中：

```
test/ export-ignore
```

现在，当你运行 `git archive` 来创建项目的压缩包时，那个目录不会被包括在归档中。

`export-subst`
在导出文件进行部署的时候，你可以将 `git log` 的格式化和关键字展开处理应用于被 `export-subst` 属性标记的部分文件。

举个例子，如果你想在项目中包含一个叫做 `LAST_COMMIT` 的文件，并在运行 `git archive` 的时候自动向它注入最新提交的元数据，可以像这样设置该文件：

```
$ echo 'Last commit date: $Format:%cd by %aN$' > LAST_COMMIT
$ echo "LAST_COMMIT export-subst" >> .gitattributes
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

运行 `git archive` 之后，该文件被归档后的内容会被替换成这样：

```
$ git archive HEAD | tar xCf ../deployment-testing -
$ cat ../deployment-testing/LAST_COMMIT
Last commit date: Tue Apr 21 08:38:48 2009 -0700 by Scott Chacon
```

你也可以用诸如提交信息或者任意的 git 注解进行替换，并且 git log 还能做简单的字词包装：

```
$ echo '$Format:Last commit: %h by %aN at %cd%n%+w(76,6,9)%B$' > LAST_COMMIT
$ git commit -am 'export-subst 使用 git log 的自定义格式化工具

git archive 直接使用 git log 的 `pretty=format:`
处理器，并在输出中移除两侧的 `$Format:` 和 `$`
标记。
'
$ git archive @ | tar xfO - LAST_COMMIT
Last commit: 312ccc8 by Jim Hill at Fri May 8 09:14:04 2015 -0700
       export-subst 使用 git log 的自定义格式化工具

         git archive 直接使用 git log 的 `pretty=format:` 处理器，并
         在输出中移除两侧的 `$Format:` 和 `$` 标记。
```

由此得到的归档适用于（当前的）部署工作。然而和其他的导出归档一样，它并不适用于后继的部署工作。

合并策略

通过 Git 属性，你还能对项目中的特定文件指定不同的合并策略。 一个非常有用的选项就是，告诉 Git 当特定文件发生冲突时不要尝试合并它们，而是直接使用你这边的内容。

考虑如下场景：项目中有一个分叉的或者定制过的特性分支，你希望该分支上的更改能合并回你的主干分支，同时需要忽略其中某些文件。此时这个合并策略就能派上用场。 假设你有一个数据库设置文件 `database.xml` ，在两个分支中它是不同的，而你想合并另一个分支到你的分支上，又不想弄乱该数据库文件。 你可以设置属性如下：

```
database.xml merge=ours
```

然后定义一个虚拟的合并策略，叫做 `ours`：

```
$ git config --global merge.ours.driver true
```

如果你合并了另一个分支，`database.xml` 文件不会有合并冲突，相反会显示如下信息：

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

这里，`database.xml` 保持了主干分支中的原始版本。

8.3 Git 钩子

Git 钩子

和其它版本控制系统一样，Git 能在特定的重要动作发生时触发自定义脚本。 有两组这样的钩子：客户端的和服务器端的。 客户端钩子由诸如提交和合并这样的操作所调用，而服务器端钩子作用于诸如接收被推送的提交这样的联网操作。 你可以随心所欲地运用这些钩子。

安装一个钩子

钩子都被存储在 Git 目录下的 `hooks` 子目录中。 也即绝大部分项目中的 `.git/hooks` 。 当你用 `git init` 初始化一个新版本库时，Git 默认会在这个目录中放置一些示例脚本。这些脚本除了本身可以被调用外，它们还透露了被触发时所传入的参数。 所有的示例都是 shell 脚本，其中一些还混杂了 Perl 代码，不过，任何正确命名的可执行脚本都可以正常使用 —— 你可以用 Ruby 或 Python，或其它语言编写它们。 这些示例的名字都是以 `.sample` 结尾，如果你想启用它们，得先移除这个后缀。

把一个正确命名且可执行的文件放入 Git 目录下的 `hooks` 子目录中，即可激活该钩子脚本。 这样一来，它就能被 Git 调用。 接下来，我们会讲解常用的钩子脚本类型。

客户端钩子

客户端钩子分为很多种。 下面把它们分为：提交工作流钩子、电子邮件工作流钩子和其它钩子。

> NOTE
>
> 需要注意的是，克隆某个版本库时，它的客户端钩子 并不 随同复制。 如果需要靠这些脚本来强制维持某种策略，建议你在服务器端实现这一功能。（请参照 使用强制策略的一个例子中的例子。）

提交工作流钩子

前四个钩子涉及提交的过程。

`pre-commit` 钩子在键入提交信息前运行。 它用于检查即将提交的快照，例如，检查是否有所遗漏，确保测试运行，以及核查代码。 如果该钩子以非零值退出，Git 将放弃此次提交，不过你可以用 `git commit --no-verify` 来绕过这个环节。 你可以利用该钩子，来检查代码风格是否一致（运行类似 `lint` 的程序）、尾随空白字符是否存在（自带的钩子就是这么做的），或新方法的文档是否适当。

`prepare-commit-msg` 钩子在启动提交信息编辑器之前，默认信息被创建之后运行。 它允许你编辑提交者所看到的默认信息。 该钩子接收一些选项：存有当前提交信息的文件的路径、提交类型和修补提交的提交的 SHA-1 校验。 它对一般的提交来说并没有什么用；然而对那些会自动产生默认信息的提交，如提交信息模板、合并提交、压缩提交和修订提交等非常实用。 你可以结合提交模板来使用它，动态地插入信息。

`commit-msg` 钩子接收一个参数，此参数即上文提到的，存有当前提交信息的临时文件的路径。 如果该钩子脚本以非零值退出，Git 将放弃提交，因此，可以用来在提交通过前验证项目状态或提交信息。 在本章的最后一节，我们将展示如何使用该钩子来核对提交信息是否遵循指定的模板。

`post-commit` 钩子在整个提交过程完成后运行。 它不接收任何参数，但你可以很容易地通过运行 `git log -1 HEAD` 来获得最后一次的提交信息。 该钩子一般用于通知之类的事情。

电子邮件工作流钩子
你可以给电子邮件工作流设置三个客户端钩子。 它们都是由 `git am` 命令调用的，因此如果你没有在你的工作流中用到这个命令，可以跳到下一节。 如果你需要通过电子邮件接收由 `git format-patch` 产生的补丁，这些钩子也许用得上。

第一个运行的钩子是 `applypatch-msg` 。 它接收单个参数：包含请求合并信息的临时文件的名字。 如果脚本返回非零值，Git 将放弃该补丁。 你可以用该脚本来确保提交信息符合格式，或直接用脚本修正格式错误。

下一个在 `git am` 运行期间被调用的是 `pre-applypatch` 。 有些难以理解的是，它正好运行于应用补丁 之后，产生提交之前，所以你可以用它在提交前检查快照。 你可以用这个脚本运行测试或检查工作区。 如果有什么遗漏，或测试未能通过，脚本会以非零值退出，中断 `git am` 的运行，这样补丁就不会被提交。

`post-applypatch` 运行于提交产生之后，是在 `git am` 运行期间最后被调用的钩子。 你可以用它把结果通知给一个小组或所拉取的补丁的作者。 但你没办法用它停止打补丁的过程。

其它客户端钩子
`pre-rebase` 钩子运行于变基之前，以非零值退出可以中止变基的过程。 你可以使用这个钩子来禁止对已经推送的提交变基。 Git 自带的 `pre-rebase` 钩子示例就是这么做的，不过它所做的一些假设可能与你的工作流程不匹配。

`post-rewrite` 钩子被那些会替换提交记录的命令调用，比如 `git commit --amend` 和 `git rebase`（不过不包括 `git filter-branch` ）。 它唯一的参数是触发重写的命令名，同时从标准输入中接受一系列重写的提交记录。 这个钩子的用途很大程度上跟 `post-checkout` 和 `post-merge` 差不多。

在 `git checkout` 成功运行后， `post-checkout` 钩子会被调用。你可以根据你的项目环境用它调整你的工作目录。 其中包括放入大的二进制文件、自动生成文档或进行其他类似这样的操作。

在 `git merge` 成功运行后， `post-merge` 钩子会被调用。 你可以用它恢复 Git 无法跟踪的工作区数据，比如权限数据。 这个钩子也可以用来验证某些在 Git 控制之外的文件是否存在，这样你就能在工作区改变时，把这些文件复制进来。

`pre-push` 钩子会在 `git push` 运行期间， 更新了远程引用但尚未传送对象时被调用。 它接受远程分支的名字和位置作为参数，同时从标准输入中读取一系列待更新的引用。 你可以在推送开始之前，用它

验证对引用的更新操作（一个非零的退出码将终止推送过程）。

Git 的一些日常操作在运行时，偶尔会调用 `git gc --auto` 进行垃圾回收。 `pre-auto-gc` 钩子会在垃圾回收开始之前被调用，可以用它来提醒你现在要回收垃圾了，或者依情形判断是否要中断回收。

服务器端钩子

除了客户端钩子，作为系统管理员，你还可以使用若干服务器端的钩子对项目强制执行各种类型的策略。这些钩子脚本在推送到服务器之前和之后运行。 推送到服务器前运行的钩子可以在任何时候以非零值退出，拒绝推送并给客户端返回错误消息，还可以依你所想设置足够复杂的推送策略。

`pre-receive`

处理来自客户端的推送操作时，最先被调用的脚本是 `pre-receive`。 它从标准输入获取一系列被推送的引用。如果它以非零值退出，所有的推送内容都不会被接受。 你可以用这个钩子阻止对引用进行非快进（non-fast-forward）的更新，或者对该推送所修改的所有引用和文件进行访问控制。

`update`

`update` 脚本和 `pre-receive` 脚本十分类似，不同之处在于它会为每一个准备更新的分支各运行一次。 假如推送者同时向多个分支推送内容，`pre-receive` 只运行一次，相比之下 `update` 则会为每一个被推送的分支各运行一次。 它不会从标准输入读取内容，而是接受三个参数：引用的名字（分支），推送前的引用指向的内容的 SHA-1 值，以及用户准备推送的内容的 SHA-1 值。 如果 update 脚本以非零值退出，只有相应的那一个引用会被拒绝；其余的依然会被更新。

`post-receive`

`post-receive` 挂钩在整个过程完结以后运行，可以用来更新其他系统服务或者通知用户。 它接受与 `pre-receive` 相同的标准输入数据。 它的用途包括给某个邮件列表发信，通知持续集成（continous integration）的服务器，或者更新问题追踪系统（ticket-tracking system） —— 甚至可以通过分析提交信息来决定某个问题（ticket）是否应该被开启，修改或者关闭。 该脚本无法终止推送进程，不过客户端在它结束运行之前将保持连接状态，所以如果你想做其他操作需谨慎使用它，因为它将耗费你很长的一段时间。

8.4 使用强制策略的一个例子

使用强制策略的一个例子

在本节中，你将应用前面学到的知识建立这样一个 Git 工作流程：检查提交信息的格式，并且指定只能由特定用户修改项目中特定的子目录。 你将编写一个客户端脚本来提示开发人员他们的推送是否会被拒绝，以及一个服务器端脚本来实际执行这些策略。

我们待会展示的脚本是用 Ruby 写的，部分是由于我习惯用它写脚本，另外也因为 Ruby 简单易懂，即便你没写过它也能看明白。 不过任何其他语言也一样适用。所有 Git 自带的示例钩子脚本都是用 Perl 或 Bash 写的，所以你能从它们中找到相当多的这两种语言的钩子示例。

服务器端钩子

所有服务器端的工作都将在你的 `hooks` 目录下的 `update` 脚本中完成。 `update` 脚本会为每一个提交的分支各运行一次，它接受三个参数：

- 被推送的引用的名字
- 推送前分支的修订版本（revision）
- 用户准备推送的修订版本（revision）

如果推送是通过 SSH 进行的，还可以获知进行此次推送的用户的信息。 如果你允许所有操作都通过公匙授权的单一帐号（比如“git”）进行，就有必要通过一个 shell 包装脚本依据公匙来判断用户的身份，并且相应地设定环境变量来表示该用户的身份。 下面就假设 `$USER` 环境变量里存储了当前连接的用户的身份，你的 update 脚本首先搜集一切需要的信息：

```
#!/usr/bin/env ruby

$refname = ARGV[0]
$oldrev  = ARGV[1]
$newrev  = ARGV[2]
$user    = ENV['USER']

puts "Enforcing Policies..."
puts "(#{$refname}) (#{$oldrev[0,6]}) (#{$newrev[0,6]})"
```

是的，我们这里用的都是全局变量。 请勿在此吐槽——这样做只是为了方便展示而已。

指定特殊的提交信息格式

你的第一项任务是要求每一条提交信息都必须遵循某种特殊的格式。 作为目标，假定每一条信息必须包含一条形似“ref: 1234”的字符串，因为你想把每一次提交对应到问题追踪系统（ticketing system）中的某个事项。 你要逐一检查每一条推送上来的提交内容，看看提交信息是否包含这么一个字符串，然后，如果某个提交里不包含这个字符串，以非零返回值退出从而拒绝此次推送。

把 `$newrev` 和 `$oldrev` 变量的值传给一个叫做 `git rev-list` 的 Git 底层命令，你可以获取所有提交的 SHA-1 值列表。 `git rev-list` 基本类似 `git log` 命令，但它默认只输出 SHA-1 值而已，没有其他信息。 所以要获取由一次提交到另一次提交之间的所有 SHA-1 值，可以像这样运行：

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

你可以截取这些输出内容，循环遍历其中每一个 SHA-1 值，找出与之对应的提交信息，然后用正则表达式来测试该信息包含的内容。

下一步要实现从每个提交中提取出提交信息。 使用另一个叫做 `git cat-file` 的底层命令来获得原始的提交数据。 我们将在 Git 内部原理 了解到这些底层命令的细节；现在暂时先看一下这条命令的输出：

```
$ git cat-file commit ca82a6
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

通过 SHA-1 值获得提交中的提交信息的一个简单办法是找到提交的第一个空行，然后取从它往后的所有内容。 可以使用 Unix 系统的 `sed` 命令来实现该效果：

```
$ git cat-file commit ca82a6 | sed '1,/^$/d'
changed the version number
```

你可以用这条咒语从每一个待推送的提交里提取提交信息，然后在提取的内容不符合要求时退出。 为了退出脚本和拒绝此次推送，返回非零值。 整个脚本大致如下：

```
$regex = /\[ref: (\d+)\]/

# 指定自定义的提交信息格式
def check_message_format
  missed_revs = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  missed_revs.each do |rev|
    message = `git cat-file commit #{rev} | sed '1,/^$/d'`
    if !$regex.match(message)
      puts "[POLICY] Your message is not formatted correctly"
      exit 1
    end
  end
end
check_message_format
```

把这一段放在 `update` 脚本里，所有包含不符合指定规则的提交都会遭到拒绝。

指定基于用户的访问权限控制列表（ACL）系统
假设你需要添加一个使用访问权限控制列表的机制，来指定哪些用户对项目的哪些部分有推送权限。 某些用户具有全部的访问权，其他人只对某些子目录或者特定的文件具有推送权限。 为了实现这一点，你要把相关的规则写入位于服务器原始 Git 仓库的 acl 文件中。 你还需要让 `update` 钩子检阅这些规则，审视推送的提交内容中被修改的所有文件，然后决定执行推送的用户是否对所有这些文件都有权限。

先从写一个 ACL 文件开始吧。 这里使用的格式和 CVS 的 ACL 机制十分类似：它由若干行构成，第一项内容是 `avail` 或者 `unavail`，接着是逗号分隔的适用该规则的用户列表，最后一项是适用该规则的路径（该项空缺表示没有路径限制）。 各项由管道符 `|` 隔开。

在本例中，你会有几个管理员，一些对 `doc` 目录具有权限的文档作者，以及一位仅对 `lib` 和 `tests` 目录具有权限的开发人员，相应的 ACL 文件如下：

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

首先把这些数据读入你要用到的数据结构里。 在本例中，为保持简洁，我们暂时只实现 `avail` 的规则。 下面这个方法生成一个关联数组，它的键是用户名，值是一个由该用户有写权限的所有目录组成的数组：

```
def get_acl_access_data(acl_file)
  # 读取ACL数据
  acl_file = File.read(acl_file).split("\n").reject { |line| line == '' }
  access = {}
  acl_file.each do |line|
    avail, users, path = line.split('|')
    next unless avail == 'avail'
    users.split(',').each do |user|
      access[user] ||= []
      access[user] << path
    end
  end
  access
end
```

对于之前给出的 ACL 规则文件，这个 `get_acl_access_data` 方法返回的数据结构如下：

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
```

```
  "ebronte"=>["doc"]}
```

既然拿到了用户权限的数据，接下来你需要找出提交都修改了哪些路径，从而才能保证推送者对所有这些路径都有权限。

使用 `git log` 的 `--name-only` 选项（在第二章里简单地提过），我们可以轻而易举的找出一次提交里修改的文件：

```
$ git log -1 --name-only --pretty=format:'' 9f585d

README
lib/test.rb
```

使用 `get_acl_access_data` 返回的 ACL 结构来一一核对每次提交修改的文件列表，就能找出该用户是否有权限推送所有的提交内容:

```
# 仅允许特定用户修改项目中的特定子目录
def check_directory_perms
  access = get_acl_access_data('acl')

  # 检查是否有人在向他没有权限的地方推送内容
  new_commits = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  new_commits.each do |rev|
    files_modified = `git log -1 --name-only --pretty=format:'' #{rev}`.split("\n")
    files_modified.each do |path|
      next if path.size == 0
      has_file_access = false
      access[$user].each do |access_path|
        if !access_path  # 用户拥有完全访问权限
           || (path.start_with? access_path) # 或者对此路径有访问权限
          has_file_access = true
        end
      end
      if !has_file_access
        puts "[POLICY] You do not have access to push to #{path}"
        exit 1
      end
    end
  end
end

check_directory_perms
```

通过 `git rev-list` 获取推送到服务器的所有提交。 接着，对于每一个提交，找出它修改的文件，然后确保推送者具有这些文件的推送权限。

现在你的用户没法推送带有不正确的提交信息的内容，也不能在准许他们访问范围之外的位置做出修改。

测试一下
如果已经把上面的代码放到 `.git/hooks/update` 文件里了，运行

`chmod u+x .git/hooks/update`，然后尝试推送一个不符合格式的提交，你会得到以下的提示：

```
$ git push -f origin master
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

这里有几个有趣的信息。 首先，我们可以看到钩子运行的起点。

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

注意这是从 update 脚本开头输出到标准输出的。 所有从脚本输出到标准输出的内容都会转发给客户端。

下一个值得注意的部分是错误信息。

```
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

第一行是我们的脚本输出的，剩下两行是 Git 在告诉我们 update 脚本退出时返回了非零值因而推送遭到了拒绝。 最后一点：

```
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

你会看到每个被你的钩子拒之门外的引用都收到了一个 remote rejected 信息，它告诉你正是钩子无法成功运行导致了推送的拒绝。

又或者某人想修改一个自己不具备权限的文件然后推送了一个包含它的提交，他将看到类似的提示。 比如，一个文档作者尝试推送一个修改到 `lib` 目录的提交，他会看到

```
[POLICY] You do not have access to push to lib/test.rb
```

从今以后，只要 `update` 脚本存在并且可执行，我们的版本库中永远都不会包含不符合格式的提交信息，并且用户都会待在沙箱里面。

客户端钩子

这种方法的缺点在于，用户推送的提交遭到拒绝后无法避免的抱怨。 辛辛苦苦写成的代码在最后时刻惨遭拒绝是十分让人沮丧且具有迷惑性的；更可怜的是他们不得不修改提交历史来解决问题，这个方法并不能让每一个人满意。

逃离这种两难境地的法宝是给用户一些客户端的钩子，在他们犯错的时候给以警告。 然后呢，用户们就能趁问题尚未变得更难修复，在提交前消除这个隐患。 由于钩子本身不跟随克隆的项目副本分发，所以你必须通过其他途径把这些钩子分发到用户的 `.git/hooks` 目录并设为可执行文件。 虽然你可以在相同或单独的项目里加入并分发这些钩子，但是 Git 不会自动替你设置它。

首先，你应该在每次提交前核查你的提交信息，这样才能确保服务器不会因为不合条件的提交信息而拒绝你的更改。 为了达到这个目的，你可以增加 `commit-msg` 钩子。 如果你使用该钩子来读取作为第一个参数传递的提交信息，然后与规定的格式作比较，你就可以使 Git 在提交信息格式不对的情况下拒绝提交。

```
#!/usr/bin/env ruby
message_file = ARGV[0]
message = File.read(message_file)

$regex = /\[ref: (\d+)\]/

if !$regex.match(message)
  puts "[POLICY] Your message is not formatted correctly"
  exit 1
end
```

如果这个脚本位于正确的位置（`.git/hooks/commit-msg`）并且是可执行的，你提交信息的格式又是不正确的，你会看到：

```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

在这个示例中，提交没有成功。 然而如果你的提交注释信息是符合要求的，Git 会允许你提交：

```
$ git commit -am 'test [ref: 132]'
[master e05c914] test [ref: 132]
 1 file changed, 1 insertions(+), 0 deletions(-)
```

接下来我们要保证没有修改到 ACL 允许范围之外的文件。 假如你的 `.git` 目录下有前面使用过的那份 ACL 文件，那么以下的 `pre-commit` 脚本将把里面的规定执行起来：

```
#!/usr/bin/env ruby

$user    = ENV['USER']

# [ 插入上文中的 get_acl_access_data 方法 ]

# 仅允许特定用户修改项目中的特定子目录
def check_directory_perms
  access = get_acl_access_data('.git/acl')

  files_modified = `git diff-index --cached --name-only HEAD`.split("\n")
  files_modified.each do |path|
    next if path.size == 0
    has_file_access = false
    access[$user].each do |access_path|
    if !access_path || (path.index(access_path) == 0)
      has_file_access = true
    end
    if !has_file_access
      puts "[POLICY] You do not have access to push to #{path}"
      exit 1
    end
  end
end

check_directory_perms
```

这和服务器端的脚本几乎一样，除了两个重要区别。 第一，ACL 文件的位置不同，因为这个脚本在当前工作目录运行，而非 `.git` 目录。 ACL 文件的路径必须从

```
access = get_acl_access_data('acl')
```

修改成：

```
access = get_acl_access_data('.git/acl')
```

另一个重要区别是获取被修改文件列表的方式。 在服务器端的时候使用了查看提交纪录的方式，可是目前的提交都还没被记录下来呢，所以这个列表只能从暂存区域获取。 和原来的

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

不同，现在要用

```
files_modified = `git diff-index --cached --name-only HEAD`
```

不同的就只有这两个——除此之外，该脚本完全相同。 有一点要注意的是，它假定在本地运行的用户和推送到远程服务器端的相同。 如果这二者不一样，则需要手动设置一下 `$user` 变量。

在这里，我们还可以确保推送内容中不包含非快进（non-fast-forward）的引用。 出现一个不是快进（fast-forward）的引用有两种情形，要么是在某个已经推送过的提交上作变基，要么是从本地推送一个错误的分支到远程分支上。

假定为了执行这个策略，你已经在服务器上配置好了 `receive.denyDeletes` 和 `receive.denyNonFastForwards`，因而唯一还需要避免的是在某个已经推送过的提交上作变基。

下面是一个检查这个问题的 `pre-rebase` 脚本示例。 它获取所有待重写的提交的列表，然后检查它们是否存在于远程引用中。 一旦发现其中一个提交是在某个远程引用中可达的（reachable），它就终止此次变基：

```
#!/usr/bin/env ruby

base_branch = ARGV[0]
if ARGV[1]
  topic_branch = ARGV[1]
else
  topic_branch = "HEAD"
end

target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")
remote_refs = `git branch -r`.split("\n").map { |r| r.strip }

target_shas.each do |sha|
  remote_refs.each do |remote_ref|
    shas_pushed = `git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
    if shas_pushed.split("\n").include?(sha)
      puts "[POLICY] Commit #{sha} has already been pushed to #{remote_ref}"
      exit 1
    end
  end
end
```

这个脚本利用了一个第六章“修订版本选择”一节中不曾提到的语法。通过运行这个命令可以获得一系列之前推送过的提交：

```
`git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
```

`SHA^@` 语法会被解析成该提交的所有父提交。 该命令会列出在远程分支最新的提交中可达的，却在所有我们尝试推送的提交的 SHA-1 值的所有父提交中不可达的提交——也就是快进的提交。

这个解决方案主要的问题在于它有可能很慢而且常常没有必要——只要你不用 `-f` 来强制推送，服务器就会自动给出警告并且拒绝接受推送。 然而，这是个不错的练习，而且理论上能帮助你避免一次以后可能不得不回头修补的变基。

8.5 总结

总结

我们已经阐述了大部分通过自定义 Git 客户端和服务端来适应自己工作流程和项目内容的方式。 你已经学到各种各样的设置项、基于文件的选项和事件钩子，还建立了一个示例用的强制策略服务器。 无论创造出了什么样的工作流程，你都能使 Git 与它珠联璧合。

9. Git 与其他系统

第9章. Git 与其他系统

现实并不总是尽如人意。通常，你不能立刻就把接触到的每一个项目都切换到 Git。有时候你被困在使用其他 VCS 的项目中，却希望使用 Git。在本章的第一部分我们将会了解到，怎样在你的那些托管在不同系统的项目上使用 Git 客户端。

在某些时候，你可能想要将已有项目转换到 Git。本章的第二部分涵盖了从几个特定系统将你的项目迁移至 Git 的方法，即使没有预先构建好的导入工具，我们也有办法手动导入。

9.1 作为客户端的 Git

作为客户端的 Git

Git 为开发者提供了如此优秀的体验，许多人已经找到了在他们的工作站上使用 Git 的方法，即使他们团队其余的人使用的是完全不同的 VCS。 有许多这种可用的适配器，它们被叫做 “桥接”。 下面我们将要介绍几个很可能会在实际中用到的桥接。

Git 与 Subversion

很大一部分开源项目与相当多的企业项目使用 Subversion 来管理它们的源代码。 而且在大多数时间里，它已经是开源项目VCS选择的 事实标准。 它在很多方面都与曾经是源代码管理世界的大人物的 CVS 相似。

Git 中最棒的特性就是有一个与 Subversion 的双向桥接，它被称作 `git svn`。 这个工具允许你使用 Git 作为连接到 Subversion 有效的客户端，这样你可以使用 Git 所有本地的功能然后如同正在本地使用 Subversion 一样推送到 Subversion 服务器。 这意味着你可以在本地做新建分支与合并分支、使用暂存区、使用变基与拣选等等的事情，同时协作者还在继续使用他们黑暗又古老的方式。 当你试图游说公司将基础设施修改为完全支持 Git 的过程中，一个好方法是将 Git 偷偷带入到公司环境，并帮助周围的开发者提升效率。 Subversion 桥接就是进入 DVCS 世界的诱饵。

`git svn`

在 Git 中所有 Subversion 桥接命令的基础命令是 `git svn`。 它可以跟很多命令，所以我们会通过几个简单的工作流程来为你演示最常用的命令。

需要特别注意的是当你使用 `git svn` 时，就是在与 Subversion 打交道，一个与 Git 完全不同的系统。 尽管 可以 在本地新建分支与合并分支，但是你最好还是通过变基你的工作来保证你的历史尽可能是直线，并且避免做类似同时与 Git 远程服务器交互的事情。

不要重写你的历史然后尝试再次推送，同时也不要推送到一个平行的 Git 仓库来与其他使用 Git 的开发者协作。 Subversion 只能有一个线性的历史，弄乱它很容易。 如果你在一个团队中工作，其中有一些人使用 SVN 而另一些人使用 Git，你需要确保每个人都使用 SVN 服务器来协作 - 这样做会省去很多麻烦。

设置

为了演示这个功能，需要一个有写入权限的典型 SVN 仓库。 如果想要拷贝这些例子，你必须获得一份我的测试仓库的可写拷贝。 为了轻松地拷贝，可以使用 Subversion 自带的一个名为 `svnsync` 的工具。 为了这些测试，我们在 Google Code 上创建了一个 `protobuf` 项目部分拷贝的新 Subversion 仓库。 `protobuf` 是一个将结构性数据编码用于网络传输的工具。

接下来，你需要先创建一个新的本地 Subversion 仓库：

```
$ mkdir /tmp/test-svn
```

```
$ svnadmin create /tmp/test-svn
```

然后，允许所有用户改变版本属性 - 最容易的方式是添加一个返回值为 0 的 `pre-revprop-change` 脚本。

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

现在可以调用加入目标与来源仓库参数的 `svnsync init` 命令同步这个项目到本地的机器。

```
$ svnsync init file:///tmp/test-svn \
  http://progit-example.googlecode.com/svn/
```

这样就设置好了同步所使用的属性。 可以通过运行下面的命令来克隆代码：

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Transmitting file data .............................[...]
Committed revision 2.
Copied properties for revision 2.
[…]
```

虽然这个操作可能只会花费几分钟，但如果你尝试拷贝原始的仓库到另一个非本地的远程仓库时，即使只有不到 100 个的提交，这个过程也可能会花费将近一个小时。 Subversion 必须一次复制一个版本然后推送回另一个仓库 - 这低效得可笑，但却是做这件事唯一简单的方式。

开始

既然已经有了一个有写入权限的 Subversion 仓库，那么你可以开始一个典型的工作流程。 可以从 `git svn clone` 命令开始，它会将整个 Subversion 仓库导入到一个本地 Git 仓库。 需要牢记的一点是如果是从一个真正托管的 Subversion 仓库中导入，需要将 `file:///tmp/test-svn` 替换为你的 Subversion 仓库的 URL：

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /private/tmp/progit/test-svn/.git/
r1 = dcbfb5891860124cc2e8cc616cded42624897125 (refs/remotes/origin/trunk)
    A   m4/acx_pthread.m4
    A   m4/stl_hash.m4
    A   java/src/test/java/com/google/protobuf/UnknownFieldSetTest.java
    A   java/src/test/java/com/google/protobuf/WireFormatTest.java
…
r75 = 556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae (refs/remotes/origin/trunk)
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-svn/branches/my-calc-branch
, 75
Found branch parent: (refs/remotes/origin/my-calc-branch) 556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae
```

```
Following parent with do_switch
Successfully followed parent
r76 = 0fb585761df569eaecd8146c71e58d70147460a2 (refs/remotes/origin/my-calc-branch)
Checked out HEAD:
  file:///tmp/test-svn/trunk r75
```

这相当于运行了两个命令 - `git svn init` 以及紧接着的 `git svn fetch` - 你提供的 URL 。 这会花费一些时间。 测试项目只有 75 个左右的提交并且代码库并不是很大，但是 Git 必须一次一个地检出一个版本同时单独地提交它。 对于有成百上千个提交的项目，这真的可能会花费几小时甚至几天来完成。

`-T trunk -b branches -t tags` 部分告诉 Git Subversion 仓库遵循基本的分支与标签惯例。 如果你命名了不同的主干、分支或标签，可以修改这些参数。 因为这是如此地常见，所以能用 `-s` 来替代整个这部分，这表示标准布局并且指代所有那些选项。 下面的命令是相同的：

```
$ git svn clone file:///tmp/test-svn -s
```

至此，应该得到了一个已经导入了分支与标签的有效的 Git 仓库：

```
$ git branch -a
* master
  remotes/origin/my-calc-branch
  remotes/origin/tags/2.0.2
  remotes/origin/tags/release-2.0.1
  remotes/origin/tags/release-2.0.2
  remotes/origin/tags/release-2.0.2rc1
  remotes/origin/trunk
```

注意这个工具是如何将 Subversion 标签作为远程引用来管理的。 让我们近距离看一下 Git 的底层命令 `show-ref` ：

```
$ git show-ref
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/heads/master
0fb585761df569eaecd8146c71e58d70147460a2 refs/remotes/origin/my-calc-branch
bfd2d79303166789fc73af4046651a4b35c12f0b refs/remotes/origin/tags/2.0.2
285c2b2e36e467dd4d91c8e3c0c0e1750b3fe8ca refs/remotes/origin/tags/release-2.0.1
cbda99cb45d9abcb9793db1d4f70ae562a969f1e refs/remotes/origin/tags/release-2.0.2
a9f074aa89e826d6f9d30808ce5ae3ffe711feda refs/remotes/origin/tags/release-2.0.2rc1
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/remotes/origin/trunk
```

Git 在从 Git 服务器克隆时并不这样做；下面是在刚刚克隆完成的有标签的仓库的样子：

```
$ git show-ref
c3dcbe8488c6240392e8a5d7553bbffcb0f94ef0 refs/remotes/origin/master
32ef1d1c7cc8c603ab78416262cc421b80a8c2df refs/remotes/origin/branch-1
75f703a3580a9b81ead89fe1138e6da858c5ba18 refs/remotes/origin/branch-2
23f8588dde934e8f33c263c6d8359b2ae095f863 refs/tags/v0.1.0
7064938bd5e7ef47bfd79a685a62c1e2649e2ce7 refs/tags/v0.2.0
6dcb09b5b57875f334f61aebed695e2e4193db5e refs/tags/v1.0.0
```

Git 直接将标签抓取至 `refs/tags`，而不是将它们看作分支。

提交回 Subversion
现在你有了一个工作仓库，你可以在项目上做一些改动，然后高效地使用 Git 作为 SVN 客户端将你的提交推送到上游。 一旦编辑了一个文件并提交它，你就有了一个存在于本地 Git 仓库的提交，这提交在 Subversion 服务器上并不存在：

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 4af61fd] Adding git-svn instructions to the README
 1 file changed, 5 insertions(+)
```

接下来，你需要将改动推送到上游。 注意这会怎样改变你使用 Subversion 的方式 - 你可以离线做几次提交然后一次性将它们推送到 Subversion 服务器。 要推送到一个 Subversion 服务器，运行 `git svn dcommit` 命令：

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   README.txt
Committed r77
    M   README.txt
r77 = 95e0222ba6399739834380eb10afcd73e0670bc5 (refs/remotes/origin/trunk)
No changes between 4af61fd05045e07598c553167e0f31c84fd6ffe1 and refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

这会拿走你在 Subversion 服务器代码之上所做的所有提交，针对每一个做一个 Subversion 提交，然后重写你本地的 Git 提交来包含一个唯一的标识符。 这很重要因为这意味着所有你的提交的 SHA-1 校验和都改变了。 部分由于这个原因，同时使用一个基于 Git 的项目远程版本和一个 Subversion 服务器并不是一个好主意。 如果你查看最后一次提交，有新的 `git-svn-id` 被添加：

```
$ git log -1
commit 95e0222ba6399739834380eb10afcd73e0670bc5
Author: ben <ben@0b684db3-b064-4277-89d1-21af03df0a68>
Date:   Thu Jul 24 03:08:36 2014 +0000

    Adding git-svn instructions to the README

    git-svn-id: file:///tmp/test-svn/trunk@77 0b684db3-b064-4277-89d1-21af03df0a68
```

注意你原来提交的 SHA-1 校验和原来是以 `4af61fd` 开头，而现在是以 `95e0222` 开头。 如果想要既推送到一个 Git 服务器又推送到一个 Subversion 服务器，必须先推送（ `dcommit` ）到 Subversion 服务器，因为这个操作会改变你的提交数据。

拉取新改动
如果你和其他开发者一起工作，当在某一时刻你们其中之一推送时，另一人尝试推送修改会导致冲突。 那

次修改会被拒绝直到你合并他们的工作。 在 `git svn` 中，它看起来是这样的：

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: d5837c4b461b7c0e018b49d12398769d2bfc240a and refs/remotes/origin/trunk differ, using rebase:
:100644 100644 f414c433af0fd6734428cf9d2a9fd8ba00ada145 c80b6127dd04f5fcda218730ddf3a2da4eb39138 M
        README.txt
Current branch master is up to date.
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

为了解决这种情况，可以运行 `git svn rebase`，它会从服务器拉取任何你本地还没有的改动，并将你所有的工作变基到服务器的内容之上：

```
$ git svn rebase
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: eaa029d99f87c5c822c5c29039d19111ff32ef46 and refs/remotes/origin/trunk differ, using rebase:
:100644 100644 65536c6e30d263495c17d781962cfff12422693a b34372b25ccf4945fe5658fa381b075045e7702a M
        README.txt
First, rewinding head to replay your work on top of it...
Applying: update foo
Using index info to reconstruct a base tree...
M       README.txt
Falling back to patching base and 3-way merge...
Auto-merging README.txt
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

现在，所有你的工作都已经在 Subversion 服务器的内容之上了，你就可以顺利地 `dcommit`：

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   README.txt
Committed r85
    M   README.txt
r85 = 9c29704cc0bbbed7bd58160cfb66cb9191835cd8 (refs/remotes/origin/trunk)
No changes between 5762f56732a958d6cfda681b661d2a239cc53ef5 and refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

注意，和 Git 需要你在推送前合并本地还没有的上游工作不同的是， `git svn` 只会在修改发生冲突时要求你那样做（更像是 Subversion 工作的行为）。 如果其他人推送一个文件的修改然后你推送了另一个文件的修改，你的 `dcommit` 命令会正常工作：

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   configure.ac
Committed r87
    M   autogen.sh
r86 = d8450bab8a77228a644b7dc0e95977ffc61adff7 (refs/remotes/origin/trunk)
    M   configure.ac
r87 = f3653ea40cb4e26b6281cec102e35dcba1fe17c4 (refs/remotes/origin/trunk)
W: a0253d06732169107aa020390d9fefd2b1d92806 and refs/remotes/origin/trunk differ, using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 e757b59a9439312d80d5d43bb65d4a7d0389ed6d M
        autogen.sh
First, rewinding head to replay your work on top of it...
```

记住这一点很重要，因为结果是当你推送后项目的状态并不存在于你的电脑中。 如果修改并未冲突但却是不兼容的，可能会引起一些难以诊断的问题。 这与使用 Git 服务器并不同 - 在 Git 中，可以在发布前完全测试客户端系统的状态，然而在 SVN 中，你甚至不能立即确定在提交前与提交后的状态是相同的。

你也应该运行这个命令从 Subversion 服务器上拉取修改，即使你自己并不准备提交。 可以运行 `git svn fetch` 来抓取新数据，但是 `git svn rebase` 会抓取并更新你本地的提交。

```
$ git svn rebase
    M   autogen.sh
r88 = c9c5f83c64bd755368784b444bc7a0216cc1e17b (refs/remotes/origin/trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/origin/trunk.
```

每隔一会儿运行 `git svn rebase` 确保你的代码始终是最新的。 虽然需要保证当运行这个命令时工作目录是干净的。 如果有本地的修改，在运行 `git svn rebase` 之前要么储藏你的工作要么做一次临时的提交，不然，当变基会导致合并冲突时，命令会终止。

Git 分支问题

当适应了 Git 的工作流程，你大概会想要创建特性分支，在上面做一些工作，然后将它们合并入主分支。 如果你正通过 `git svn` 推送到一个 Subversion 服务器，你可能想要把你的工作变基到一个单独的分支上，而不是将分支合并到一起。 比较喜欢变基的原因是因为 Subversion 有一个线性的历史并且无法像 Git 一样处理合并，所以 `git svn` 在将快照转换成 Subversion 提交时，只会保留第一父提交。

假设你的历史像下面这样：创建了一个 `experiment` 分支，做了两次提交，然后将它们合并回 `master`。 当 `dcommit` 时，你看到输出是这样的：

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   CHANGES.txt
Committed r89
    M   CHANGES.txt
r89 = 89d492c884ea7c834353563d5d913c6adf933981 (refs/remotes/origin/trunk)
    M   COPYING.txt
    M   INSTALL.txt
Committed r90
```

```
    M   INSTALL.txt
    M   COPYING.txt
r90 = cb522197870e61467473391799148f6721bcf9a0 (refs/remotes/origin/trunk)
No changes between 71af502c214ba13123992338569f4669877f55fd and refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

在一个合并过历史提交的分支上 `dcommit` 命令工作得很好，除了当你查看你的 Git 项目历史时，它并没有重写所有你在 `experiment` 分支上所做的任意提交 - 相反，所有这些修改显示一个单独合并提交的 SVN 版本中。

当其他人克隆那些工作时，他们只会看到一个被塞入了所有改动的合并提交，就像运行了 `git merge --squash` ；他们无法看到修改从哪来或何时提交的信息。

Subversion 分支
在 Subversion 中新建分支与在 Git 中新建分支并不相同；如果你能不用它，那最好就不要用。 然而，你可以使用 git svn 在 Subversion 中创建分支并在分支上做提交。

创建一个新的 SVN 分支
要在 Subversion 中创建一个新分支，运行 `git svn branch [branchname]` ：

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r90 to file:///tmp/test-svn/branches/opera...
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-svn/branches/opera, 90
Found branch parent: (refs/remotes/origin/opera) cb522197870e61467473391799148f6721bcf9a0
Following parent with do_switch
Successfully followed parent
r91 = f1b64a3855d3c8dd84ee0ef10fa89d27f1584302 (refs/remotes/origin/opera)
```

这与 Subversion 中的 `svn copy trunk branches/opera` 命令作用相同并且是在 Subversion 服务器中操作。 需要重点注意的是它并不会检出到那个分支；如果你在这时提交，提交会进入服务器的 `trunk` 分支，而不是 `opera` 分支。

切换活动分支
Git 通过查找在历史中 Subversion 分支的头部来指出你的提交将会到哪一个分支 - 应该只有一个，并且它应该是在当前分支历史中最后一个有 `git-svn-id` 的。

如果想要同时在不止一个分支上工作，可以通过在导入的那个分支的 Subversion 提交开始来设置本地分支 `dcommit` 到特定的 Subversion 分支。 如果想要一个可以单独在上面工作的 `opera` 分支，可以运行

```
$ git branch opera remotes/origin/opera
```

现在，如果想要将你的 `opera` 分支合并入 `trunk` （你的 `master` 分支），可以用一个正常的 `git merge` 来这样做。 但是你需要通过 `-m` 来提供一个描述性的提交信息，否则合并信息会是没有用

的 “Merge branch opera”。

记住尽管使用的是 `git merge` 来做这个操作，而且合并可能会比在 Subversion 中更容易一些（因为 Git 会为你自动地检测合适的合并基础），但这并不是一个普通的 Git 合并提交。 你不得不将这个数据推送回一个 Subversion 服务器，Subversion 服务器不支持那些跟踪多个父结点的提交；所以，当推送完成后，它看起来会是一个将其他分支的所有提交压缩在一起的单独提交。 在合并一个分支到另一个分支后，你并不能像 Git 中那样轻松地回到原来的分支继续工作。 你运行的 `dcommit` 命令会将哪个分支被合并进来的信息抹掉，所以后续的合并基础计算会是错的 - dcommit 会使你的 `git merge` 结果看起来像是运行了 `git merge --squash`。 不幸的是，没有一个好的方式来避免这种情形 - Subversion 无法存储这个信息，所以当使用它做为服务器时你总是会被它的限制打垮。 为了避免这些问题，应该在合并到主干后删除本地分支（本例中是 `opera`）。

Subversion 命令
`git svn` 工具集通过提供很多功能与 Subversion 中那些相似的命令来帮助简化转移到 Git 的过程。 下面是一些提供了 Subversion 中常用功能的命令。

SVN 风格历史
如果你习惯于使用 Subversion 并且想要看 SVN 输出风格的提交历史，可以运行 `git svn log` 来查看 SVN 格式的提交历史：

```
$ git svn log
------------------------------------------------------------------------
r87 | schacon | 2014-05-02 16:07:37 -0700 (Sat, 02 May 2014) | 2 lines

autogen change

------------------------------------------------------------------------
r86 | schacon | 2014-05-02 16:00:21 -0700 (Sat, 02 May 2014) | 2 lines

Merge branch 'experiment'

------------------------------------------------------------------------
r85 | schacon | 2014-05-02 16:00:09 -0700 (Sat, 02 May 2014) | 2 lines

updated the changelog
```

关于 `git svn log`，有两件重要的事你应该知道。 首先，它是离线工作的，并不像真正的 `svn log` 命令，会向 Subversion 服务器询问数据。 其次，它只会显示已经提交到 Subversion 服务器上的提交。 还未 dcommit 的本地 Git 提交并不会显示；同样也不会显示这段时间中其他人推送到 Subversion 服务器上的提交。 它更像是最后获取到的 Subversion 服务器上的提交状态。

SVN 注解
类似 `git svn log` 命令离线模拟了 `svn log` 命令，你可以认为 `git svn blame [FILE]` 离线模拟了 `svn annotate`。 输出看起来像这样：

```
$ git svn blame README.txt
```

```
  2   temporal Protocol Buffers - Google's data interchange format
  2   temporal Copyright 2008 Google Inc.
  2   temporal http://code.google.com/apis/protocolbuffers/
  2   temporal
 22   temporal C++ Installation - Unix
 22   temporal =======================
  2   temporal
 79    schacon Committing in git-svn.
 78    schacon
  2   temporal To build and install the C++ Protocol Buffer runtime and the Protocol
  2   temporal Buffer compiler (protoc) execute the following:
  2   temporal
```

重复一次，它并不显示你在 Git 中的本地提交，也不显示同一时间被推送到 Subversion 的其他提交。

SVN 服务器信息
可以通过运行 `git svn info` 得到与 `svn info` 相同种类的信息。

```
$ git svn info
Path: .
URL: https://schacon-test.googlecode.com/svn/trunk
Repository Root: https://schacon-test.googlecode.com/svn
Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
Revision: 87
Node Kind: directory
Schedule: normal
Last Changed Author: schacon
Last Changed Rev: 87
Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)
```

这就像是在你上一次和 Subversion 服务器通讯时同步了之后，离线运行的 `blame` 与 `log` 命令。

忽略 Subversion 所忽略的
如果克隆一个在任意一处设置 `svn:ignore` 属性的 Subversion 仓库时，你也许会想要设置对应的 `.gitignore` 文件，这样就不会意外的提交那些不该提交的文件。 `git svn` 有两个命令来帮助解决这个问题。 第一个是 `git svn create-ignore`，它会为你自动地创建对应的 `.gitignore` 文件，这样你的下次提交就能包含它们。

第二个命令是 `git svn show-ignore`，它会将你需要放在 `.gitignore` 文件中的每行内容打印到标准输出，这样就可以将输出内容重定向到项目的例外文件中：

```
$ git svn show-ignore > .git/info/exclude
```

这样，你就不会由于 `.gitignore` 文件而把项目弄乱。 当你是 Subversion 团队中唯一的 Git 用户时这是一个好的选项，并且你的队友并不想要项目内存在 `.gitignore` 文件。

Git-Svn 总结
当你不得不使用 Subversion 服务器或者其他必须运行一个 Subversion 服务器的开发环境时，

`git svn` 工具很有用。 你应该把它当做一个不完全的 Git，然而，你要是不用它的话，就会在做转换的过程中遇到很多麻烦的问题。 为了不惹麻烦，尽量遵守这些准则：

- 保持一个线性的 Git 历史，其中不能有 `git merge` 生成的合并提交。 把你在主线分支外开发的全部工作变基到主线分支；而不要合并入主线分支。
- 不要建立一个单独的 Git 服务器，也不要在 Git 服务器上协作。 可以用一台 Git 服务器来帮助新来的开发者加速克隆，但是不要推送任何不包含 `git-svn-id` 条目的东西。 你可能会需要增加一个 `pre-receive` 钩子来检查每一个提交信息是否包含 `git-svn-id` 并且拒绝任何未包含的提交。

如果你遵守了那些准则，忍受用一个 Subversion 服务器来工作可以更容易些。 然而，如果有可能迁移到一个真正的 Git 服务器，那么迁移过去能使你的团队获得更多好处。

Git 与 Mercurial

DVCS 的宇宙里不只有 Git。 实际上，在这个空间里有许多其他的系统。对于如何正确地进行分布式版本管理，每一个系统都有自己的视角。 除了 Git，最流行的就是 Mercurial，并且它们两个在很多方面都很相似。

好消息是，如果你更喜欢 Git 的客户端行为但是工作在源代码由 Mercurial 控制的项目中，有一种使用 Git 作为 Mercurial 托管仓库的客户端的方法。 由于 Git 与服务器仓库是使用远程交互的，那么由远程助手实现的桥接方法就不会让人很惊讶。 这个项目的名字是 git-remote-hg，可以在https://github.com/felipec/git-remote-hg 找到。

git-remote-hg

首先，需要安装 git-remote-hg。 实际上需要将它的文件放在 PATH 变量的某个目录中，像这样：

```
$ curl -o ~/bin/git-remote-hg \
  https://raw.githubusercontent.com/felipec/git-remote-hg/master/git-remote-hg
$ chmod +x ~/bin/git-remote-hg
```

假定 `~/bin` 在 `$PATH` 变量中。 Git-remote-hg 有一个其他的依赖：`mercurial` Python 库。 如果已经安装了 Python，安装它就像这样简单：

```
$ pip install mercurial
```

（如果未安装 Python，访问 https://www.python.org/ 来获取它。 ）

需要做的最后一件事是安装 Mercurial 客户端。 如果还没有安装的话请访问 http://mercurial.selenic.com/ 来安装。

现在已经准备好摇滚了。 你所需要的一切就是一个你可以推送的 Mercurial 仓库。 很幸运，每一个 Mercurial 仓库都可以这样做，所以我们只需要使用大家用来学习 Mercurial 的“hello world”仓库就可

以了：

```
$ hg clone http://selenic.com/repo/hello /tmp/hello
```

开始
既然有一个可用的 “server-side” 仓库，我们可以通过一个典型的工作流来了解。 你将会看到，这两种系统非常相似，没有太多的出入。

和 Git 一样，首先我们克隆：

```
$ git clone hg::/tmp/hello /tmp/hello-git
$ cd /tmp/hello-git
$ git log --oneline --graph --decorate
* ac7955c (HEAD, origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/branches/default,
refs/hg/origin/bookmarks/master, master) Create a makefile
```

你会注意到与 Mercurial 仓库工作时使用了标准的 `git clone` 命令。 那是因为 git-remote-hg 工作在相当低的底层，使用类似于 Git HTTP/S 协议的机制实现的（远程助手）。 由于 Git 与 Mercurial 都设计为每一个客户端有一个仓库的完整历史，所以这个命令做了一次完整的克隆，包括所有的项目历史，并且相当快地完成。

log 命令显示了两次提交，最后一次提交指向了一大堆引用。 那说明这其中的一部分实际上并没有在那儿。 让我们看看 `.git` 目录中实际上有什么：

```
$ tree .git/refs
.git/refs
├── heads
│   └── master
├── hg
│   └── origin
│       ├── bookmarks
│       │   └── master
│       └── branches
│           └── default
├── notes
│   └── hg
├── remotes
│   └── origin
│       └── HEAD
└── tags

9 directories, 5 files
```

Git-remote-hg 尝试让结构更有 Git 风格，但是在隐藏在下面的是它管理两个轻微不同系统之间概念的映射。 `refs/hg` 目录中存储了实际的远程引用。 例如，`refs/hg/origin/branches/default` 是一个包含以“ac7955c”开始的 SHA-1 值的 Git 引用文件，是 `master` 所指向的提交。 所以 `refs/hg` 目录是一种类似 `refs/remotes/origin` 的替代品，但是它引入了书签与分支的区别。

`notes/hg` 文件是 git-remote-hg 如何在 Git 的提交散列与 Mercurial 变更集 ID 之间建立映射的起点。 让我们来探索一下：

```
$ cat notes/hg
d4c10386...

$ git cat-file -p d4c10386...
tree 1781c96...
author remote-hg <> 1408066400 -0800
committer remote-hg <> 1408066400 -0800

Notes for master

$ git ls-tree 1781c96...
100644 blob ac9117f...  65bb417...
100644 blob 485e178...  ac7955c...

$ git cat-file -p ac9117f
0a04b987be5ae354b710cefeba0e2d9de7ad41a9
```

所以 `refs/notes/hg` 指向了一个树，即在 Git 对象数据库中的一个有其他对象名字的列表。 `git ls-tree` 输出 tree 对象中所有项目的模式、类型、对象哈希与文件名。 如果深入挖掘 tree 对象中的一个项目，我们会发现在其中是一个名字为 “ac9117f” 的 blob 对象（ `master` 所指向提交的 SHA-1 散列值），包含内容 “0a04b98”（是 `default` 分支指向的 Mercurial 变更集的 ID）。

好消息是大多数情况下我们不需要关心以上这些。 典型的工作流程与使用 Git 远程仓库并没有什么不同。

在我们继续之前，这里还有一件需要注意的事情：忽略。 Mercurial 与 Git 使用非常类似的机制实现这个功能，但是一般来说你不会想要把一个 `.gitignore` 文件提交到 Mercurial 仓库中。 幸运的是，Git 有一种方式可以忽略本地磁盘仓库的文件，而且 Mercurial 格式是与 Git 兼容的，所以你只需将这个文件拷贝过去：

```
$ cp .hgignore .git/info/exclude
```

`.git/info/exclude` 文件的作用像是一个 `.gitignore` ，但是它不包含在提交中。

工作流程
假设我们已经做了一些工作并且在 `master` 分支做了几次提交，而且已经准备将它们推送到远程仓库。 这是我们仓库现在的样子：

```
$ git log --oneline --graph --decorate
* ba04a2a (HEAD, master) Update makefile
* d25d16f Goodbye
* ac7955c (origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/branches/default, refs/h
g/origin/bookmarks/master) Create a makefile
```

我们的 `master` 分支领先 `origin/master` 分支两个提交，但是那两个提交只存在于我们的本地机器

中。 让我们看看在同一时间有没有其他人做过什么重要的工作：

```
$ git fetch
From hg::/tmp/hello
   ac7955c..df85e87  master     -> origin/master
   ac7955c..df85e87  branches/default -> origin/branches/default
$ git log --oneline --graph --decorate --all
* 7b07969 (refs/notes/hg) Notes for default
* d4c1038 Notes for master
* df85e87 (origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/branches/default, refs/h
g/origin/bookmarks/master) Add some documentation
| * ba04a2a (HEAD, master) Update makefile
| * d25d16f Goodbye
|/
```

因为使用了 `--all` 标记，我们看到被 git-remote-hg 内部使用的 “notes” 引用，但是可以忽略它们。 剩下的部分是我们期望的； `origin/master` 已经前进了一次提交，同时我们的历史现在分叉了。 Mercurial 和我们本章中讨论的其他系统不一样，它能够处理合并，所以我们不需要做任何其他事情。

```
$ git merge origin/master
Auto-merging hello.c
Merge made by the 'recursive' strategy.
 hello.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git log --oneline --graph --decorate
*   0c64627 (HEAD, master) Merge remote-tracking branch 'origin/master'
|\
| * df85e87 (origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/branches/default, refs
/hg/origin/bookmarks/master) Add some documentation
* | ba04a2a Update makefile
* | d25d16f Goodbye
|/
* ac7955c Create a makefile
```

完美。 运行测试然后所有测试都通过了，所以我们准备将工作共享给团队的其他成员。

```
$ git push
To hg::/tmp/hello
   df85e87..0c64627  master -> master
```

就是这样！ 如果你现在查看一下 Mercurial 仓库，你会发现这样实现了我们所期望的：

```
$ hg log -G --style compact
o    5[tip]:4,2   dc8fa4f932b8   2014-08-14 19:33 -0700   ben
|\     Merge remote-tracking branch 'origin/master'
| |
| o  4   64f27bcefc35   2014-08-14 19:27 -0700   ben
| |    Update makefile
| |
| o  3:1   4256fc29598f   2014-08-14 19:27 -0700   ben
| |    Goodbye
| |
```

```
@ |  2   7db0b4848b3c   2014-08-14 19:30 -0700   ben
|/     Add some documentation
|
o  1   82e55d328c8c   2005-08-26 01:21 -0700   mpm
|    Create a makefile
|
o  0   0a04b987be5a   2005-08-26 01:20 -0700   mpm
     Create a standard "hello, world" program
```

序号 2 的变更集是由 Mercurial 生成的，序号 3 与序号 4 的变更集是由 git-remote-hg 生成的，通过 Git 推送上来的提交。

分支与书签
Git 只有一种类型的分支：当提交生成时移动的一个引用。 在 Mercurial 中，这种类型的引用叫作“bookmark”，它的行为非常类似于 Git 分支。

Mercurial 的 “branch” 概念则更重量级一些。 变更集生成时的分支会记录 在变更集中，意味着它会永远地存在于仓库历史中。 这个例子描述了一个在 `develop` 分支上的提交：

```
$ hg log -l 1
changeset:   6:8f65e5e02793
branch:      develop
tag:         tip
user:        Ben Straub <ben@straub.cc>
date:        Thu Aug 14 20:06:38 2014 -0700
summary:     More documentation
```

注意开头为 “branch” 的那行。 Git 无法真正地模拟这种行为（并且也不需要这样做；两种类型的分支都可以表达为 Git 的一个引用），但是 git-remote-hg 需要了解其中的区别，因为 Mercurial 关心。

创建 Mercurial 书签与创建 Git 分支一样容易。 在 Git 这边：

```
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ git push origin featureA
To hg::/tmp/hello
 * [new branch]      featureA -> featureA
```

这就是所要做的全部。 在 Mercurial 这边，它看起来像这样：

```
$ hg bookmarks
   featureA                 5:bd5ac26f11f9
$ hg log --style compact -G
@  6[tip]   8f65e5e02793   2014-08-14 20:06 -0700   ben
|    More documentation
|
o    5[featureA]:4,2   bd5ac26f11f9   2014-08-14 20:02 -0700   ben
|\     Merge remote-tracking branch 'origin/master'
| |
| o  4   0434aaa6b91f   2014-08-14 20:01 -0700   ben
```

```
| |    update makefile
| |
| o  3:1   318914536c86   2014-08-14 20:00 -0700   ben
| |    goodbye
| |
o |  2   f098c7f45c4f   2014-08-14 20:01 -0700   ben
|/     Add some documentation
|
o  1   82e55d328c8c   2005-08-26 01:21 -0700   mpm
|    Create a makefile
|
o  0   0a04b987be5a   2005-08-26 01:20 -0700   mpm
     Create a standard "hello, world" program
```

注意在修订版本 5 上的新 `[featureA]` 标签。 在 Git 这边这些看起来像是 Git 分支，除了一点：不能从 Git 这边删除书签（这是远程助手的一个限制）。

你也可以工作在一个 “重量级” 的 Mercurial branch：只需要在 `branches` 命名空间内创建一个分支：

```
$ git checkout -b branches/permanent
Switched to a new branch 'branches/permanent'
$ vi Makefile
$ git commit -am 'A permanent change'
$ git push origin branches/permanent
To hg::/tmp/hello
 * [new branch]      branches/permanent -> branches/permanent
```

下面是 Mercurial 这边的样子：

```
$ hg branches
permanent                      7:a4529d07aad4
develop                        6:8f65e5e02793
default                        5:bd5ac26f11f9 (inactive)
$ hg log -G
o  changeset:   7:a4529d07aad4
|  branch:      permanent
|  tag:         tip
|  parent:      5:bd5ac26f11f9
|  user:        Ben Straub <ben@straub.cc>
|  date:        Thu Aug 14 20:21:09 2014 -0700
|  summary:     A permanent change
|
| @  changeset:   6:8f65e5e02793
|/   branch:      develop
|    user:        Ben Straub <ben@straub.cc>
|    date:        Thu Aug 14 20:06:38 2014 -0700
|    summary:     More documentation
|
o    changeset:   5:bd5ac26f11f9
|\   bookmark:    featureA
| |  parent:      4:0434aaa6b91f
| |  parent:      2:f098c7f45c4f
| |  user:        Ben Straub <ben@straub.cc>
| |  date:        Thu Aug 14 20:02:21 2014 -0700
| |  summary:     Merge remote-tracking branch 'origin/master'
```

```
[...]
```

分支名字 “permanent” 记录在序号 7 的变更集中。

在 Git 这边，对于其中任何一种风格的分支的工作都是相同的：仅仅是正常做的检出、提交、抓取、合并、拉取与推送。 还有需要知道的一件事情是 Mercurial 不支持重写历史，只允许添加历史。 下面是我们的 Mercurial 仓库在交互式的变基与强制推送后的样子：

```
$ hg log --style compact -G
o  10[tip]   99611176cbc9   2014-08-14 20:21 -0700   ben
|    A permanent change
|
o  9   f23e12f939c3   2014-08-14 20:01 -0700   ben
|    Add some documentation
|
o  8:1   c16971d33922   2014-08-14 20:00 -0700   ben
|    goodbye
|
| o  7:5   a4529d07aad4   2014-08-14 20:21 -0700   ben
| |    A permanent change
| |
| | @  6   8f65e5e02793   2014-08-14 20:06 -0700   ben
| |/     More documentation
| |
| o    5[featureA]:4,2   bd5ac26f11f9   2014-08-14 20:02 -0700   ben
| |\     Merge remote-tracking branch 'origin/master'
| | |
| | o  4   0434aaa6b91f   2014-08-14 20:01 -0700   ben
| | |    update makefile
| | |
+---o  3:1   318914536c86   2014-08-14 20:00 -0700   ben
| |      goodbye
| |
| o  2   f098c7f45c4f   2014-08-14 20:01 -0700   ben
|/     Add some documentation
|
o  1   82e55d328c8c   2005-08-26 01:21 -0700   mpm
|    Create a makefile
|
o  0   0a04b987be5a   2005-08-26 01:20 -0700   mpm
     Create a standard "hello, world" program
```

变更集 8、9 与 10 已经被创建出来并且属于 `permanent` 分支，但是旧的变更集依然在那里。 这会让使用 Mercurial 的团队成员非常困惑，所以要避免这种行为。

Mercurial 总结
Git 与 Mercurial 如此相似，以至于跨这两个系统进行工作十分流畅。 如果能注意避免改变在你机器上的历史（就像通常建议的那样），你甚至并不会察觉到另一端是 Mercurial。

Git 与 Perforce

在企业环境中 Perforce 是非常流行的版本管理系统。 它大概起始于 1995 年，这使它成为了本章中介绍的最古老的系统。 就其本身而言，它设计时带有当时时代的局限性；它假定你始终连接到一个单独的中央

服务器，本地磁盘只保存一个版本。 诚然，它的功能与限制适合几个特定的问题，但实际上，在很多情况下，将使用 Perforce 的项目换做使用 Git 会更好。

如果你决定混合使用 Perforce 与 Git 这里有两种选择。 第一个我们要介绍的是 Perforce 官方制作的“Git Fusion” 桥接，它可以将 Perforce 仓库中的子树表示为一个可读写的 Git 仓库。 第二个是 git-p4，一个客户端桥接允许你将 Git 作为 Perforce 的客户端使用，而不用在 Perforce 服务器上做任何重新的配置。

Git Fusion

Perforce 提供了一个叫作 Git Fusion 的产品（可在 http://www.perforce.com/git-fusion 获得），它将会在服务器这边同步 Perforce 服务器与 Git 仓库。

设置

针对我们的例子，我们将会使用最简单的方式安装 Git Fusion：下载一个虚拟机来运行 Perforce 守护进程与 Git Fusion。 可以从 http://www.perforce.com/downloads/Perforce/20-User 获得虚拟机镜像，下载完成后将它导入到你最爱的虚拟机软件中（我们将会使用 VirtualBox）。

在第一次启动机器后，它会询问你自定义三个 Linux 用户（`root`、`perforce` 与 `git`）的密码，并且提供一个实例名字来区分在同一网络下不同的安装。 当那些都完成后，将会看到这样：

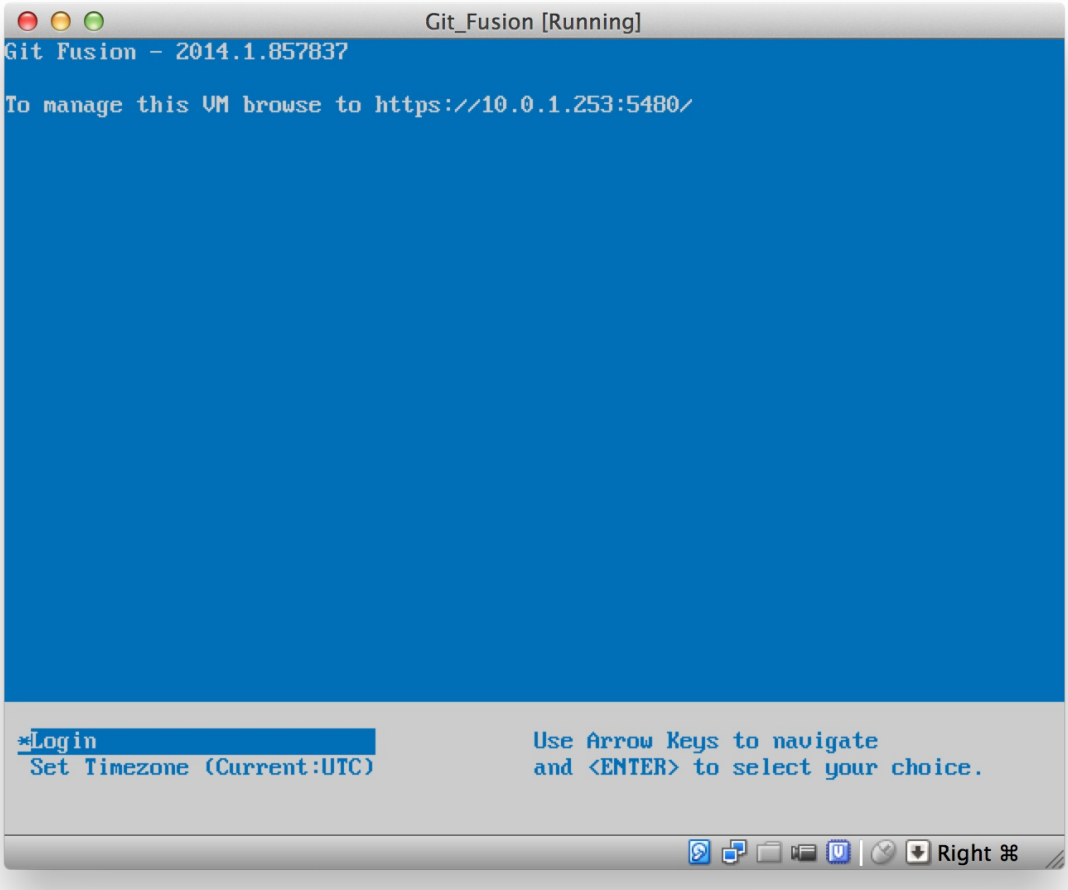


Figure 9-1. Git Fusion 虚拟机启动屏幕。

应当注意显示在这儿的 IP 地址，我们将会在后面用到。 接下来，我们将会创建一个 Perforce 用户。 选择底部的 “Login” 选项并按下回车（或者用 SSH 连接到这台机器），然后登录为 `root`。 然后使用这些命令创建一个用户：

```
$ p4 -p localhost:1666 -u super user -f john
$ p4 -p localhost:1666 -u john passwd
$ exit
```

第一个命令将会打开一个 VI 编辑器来自定义用户，但是可以通过输入 `:wq` 并回车来接受默认选项。 第二个命令将会提示输入密码两次。 这就是所有我们要通过终端提示符做的事情，所以现在可以退出当前会话了。

接下来要做的事就是告诉 Git 不要验证 SSL 证书。 Git Fusion 镜像内置一个证书，但是域名并不匹配你的虚拟主机的 IP 地址，所以 Git 会拒绝 HTTPS 连接。 如果要进行永久安装，查阅 Perforce Git Fusion

手册来安装一个不同的证书；然而，对于我们这个例子来说，这已经足够了。

```
$ export GIT_SSL_NO_VERIFY=true
```

现在我们可以测试所有东西是不是正常工作。

```
$ git clone https://10.0.1.254/Talkhouse
Cloning into 'Talkhouse'...
Username for 'https://10.0.1.254': john
Password for 'https://john@10.0.1.254':
remote: Counting objects: 630, done.
remote: Compressing objects: 100% (581/581), done.
remote: Total 630 (delta 172), reused 0 (delta 0)
Receiving objects: 100% (630/630), 1.22 MiB | 0 bytes/s, done.
Resolving deltas: 100% (172/172), done.
Checking connectivity... done.
```

虚拟机镜像自带一个可以克隆的样例项目。 这里我们会使用之前创建的 `john` 用户，通过 HTTPS 进行克隆；Git 询问此次连接的凭证，但是凭证缓存会允许我们跳过这步之后的任意后续请求。

Fusion 配置
一旦安装了 Git Fusion，你会想要调整配置。 使用你最爱的 Perforce 客户端做这件事实际上相当容易；只需要映射 Perforce 服务器上的 `//.git-fusion` 目录到你的工作空间。 文件结构看起来像这样：

```
$ tree
.
├── objects
│   ├── repos
│   │   └── [...]
│   └── trees
│       └── [...]
│
├── p4gf_config
├── repos
│   └── Talkhouse
│       └── p4gf_config
└── users
    └── p4gf_usermap

498 directories, 287 files
```

`objects` 目录被 Git Fusion 内部用来双向映射 Perforce 对象与 Git 对象，你不必弄乱那儿的任何东西。 在这个目录中有一个全局的 `p4gf_config` 文件，每个仓库中也会有一份 - 这些配置文件决定了 Git Fusion 的行为。 让我们看一下根目录下的文件：

```
[repo-creation]
charset = utf8

[git-to-perforce]
```

```
change-owner = author
enable-git-branch-creation = yes
enable-swarm-reviews = yes
enable-git-merge-commits = yes
enable-git-submodules = yes
preflight-commit = none
ignore-author-permissions = no
read-permission-check = none
git-merge-avoidance-after-change-num = 12107

[perforce-to-git]
http-url = none
ssh-url = none

[@features]
imports = False
chunked-push = False
matrix2 = False
parallel-push = False

[authentication]
email-case-sensitivity = no
```

这里我们并不会深入介绍这些选项的含义，但是要注意这是一个 INI 格式的文本文件，就像 Git 的配置。这个文件指定了全局选项，但它可以被仓库特定的配置文件覆盖，像是`repos/Talkhouse/p4gf_config`。 如果打开这个文件，你会看到有一些与全局默认不同设置的`[@repo]`区块。 你也会看到像下面这样的区块：

```
[Talkhouse-master]
git-branch-name = master
view = //depot/Talkhouse/main-dev/... ...
```

这是一个 Perforce 分支与一个 Git 分支的映射。 这个区块可以被命名成你喜欢的名字，只要保证名字是唯一的即可。 `git-branch-name`允许你将在 Git 下显得笨重的仓库路径转换为更友好的名字。`view`选项使用标准视图映射语法控制 Perforce 文件如何映射到 Git 仓库。 可以指定一个以上的映射，就像下面的例子：

```
[multi-project-mapping]
git-branch-name = master
view = //depot/project1/main/... project1/...
       //depot/project2/mainline/... project2/...
```

通过这种方式，如果正常工作空间映射包含对目录结构的修改，可以将其复制为一个 Git 仓库。

最后一个我们讨论的文件是`users/p4gf_usermap`，它将 Perforce 用户映射到 Git 用户，但你可能不会需要它。 当从一个 Perforce 变更集转换为一个 Git 提交时，Git Fusion 的默认行为是去查找 Perforce 用户，然后把邮箱地址与全名存储在 Git 的 author/commiter 字段中。 当反过来转换时，默认的行为是根据存储在 Git 提交中 author 字段中的邮箱地址来查找 Perforce 用户，然后以该用户提交变更集（以及权限的应用）。 大多数情况下，这个行为工作得很好，但是考虑下面的映射文件：

```
john john@example.com "John Doe"
john johnny@appleseed.net "John Doe"
bob employeeX@example.com "Anon X. Mouse"
joe employeeY@example.com "Anon Y. Mouse"
```

每一行的格式都是 `<user> <email> "<full name>"`，创建了一个单独的用户映射。 前两行映射不同的邮箱地址到同一个 Perforce 用户账户。 当使用几个不同的邮箱地址（或改变邮箱地址）生成 Git 提交并且想要让他们映射到同一个 Perforce 用户时这会很有用。 当从一个 Perforce 变更集创建一个 Git 提交时，第一个匹配 Perforce 用户的行会被用作 Git 作者信息。

最后两行从创建的 Git 提交中掩盖了 Bob 与 Joe 的真实名字与邮箱地址。 当你想要将一个内部项目开源，但不想将你的雇员目录公布到全世界时这很不错。 注意邮箱地址与全名需要是唯一的，除非想要所有的 Git 提交都属于一个虚构的作者。

工作流程
Perforce Git Fusion 是在 Perforce 与 Git 版本控制间双向的桥接。 让我们看一下在 Git 这边工作是什么样的感觉。 假定我们在 “Jam” 项目中使用上述的配置文件映射了，可以这样克隆：

```
$ git clone https://10.0.1.254/Jam
Cloning into 'Jam'...
Username for 'https://10.0.1.254': john
Password for 'https://ben@10.0.1.254':
remote: Counting objects: 2070, done.
remote: Compressing objects: 100% (1704/1704), done.
Receiving objects: 100% (2070/2070), 1.21 MiB | 0 bytes/s, done.
remote: Total 2070 (delta 1242), reused 0 (delta 0)
Resolving deltas: 100% (1242/1242), done.
Checking connectivity... done.
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/rel2.1
$ git log --oneline --decorate --graph --all
* 0a38c33 (origin/rel2.1) Create Jam 2.1 release branch.
| * d254865 (HEAD, origin/master, origin/HEAD, master) Upgrade to latest metrowerks on Beos -- the Inte
l one.
| * bd2f54a Put in fix for jam's NT handle leak.
| * c0f29e7 Fix URL in a jam doc
| * cc644ac Radstone's lynx port.
```

当首次这样做时，会花费一些时间。 这里发生的是 Git Fusion 会将在 Perforce 历史中所有合适的变更集转换为 Git 提交。 这发生在服务器端本地，所以会相当快，但是如果有很多历史，那么它还是会花费一些时间。 后来的抓取会做增量转换，所以会感觉更像 Git 的本地速度。

如你所见，我们的仓库看起来像之前使用过的任何一个 Git 仓库了。 这里有三个分支，Git 已经帮助创建了一个跟踪 `origin/master` 的本地 `master` 分支。 让我们做一些工作，创建几个新提交：

```
# ...
$ git log --oneline --decorate --graph --all
* cfd46ab (HEAD, master) Add documentation for new feature
* a730d77 Whitespace
* d254865 (origin/master, origin/HEAD) Upgrade to latest metrowerks on Beos -- the Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

我们有两个新提交。 现在我们检查下是否有其他人在工作：

```
$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://10.0.1.254/Jam
   d254865..6afeb15  master     -> origin/master
$ git log --oneline --decorate --graph --all
* 6afeb15 (origin/master, origin/HEAD) Update copyright
| * cfd46ab (HEAD, master) Add documentation for new feature
| * a730d77 Whitespace
|/
* d254865 Upgrade to latest metrowerks on Beos -- the Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

看起来有人在工作！ 从这个视图来看你并不知道这点，但是 `6afeb15` 提交确实是使用 Perforce 客户端创建的。 从 Git 的视角看它仅仅只是另一个提交，准确地说是一个点。 让我们看看 Perforce 服务器如何处理一个合并提交：

```
$ git merge origin/master
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 6), reused 0 (delta 0)
remote: Perforce: 100% (3/3) Loading commit tree into memory...
remote: Perforce: 100% (5/5) Finding child commits...
remote: Perforce: Running git fast-export...
remote: Perforce: 100% (3/3) Checking commits...
remote: Processing will continue even if connection is closed.
remote: Perforce: 100% (3/3) Copying changelists...
remote: Perforce: Submitting new Git commit objects to Perforce: 4
To https://10.0.1.254/Jam
   6afeb15..89cba2b  master -> master
```

Git 认为它成功了。 让我们从 Perforce 的视角看一下 `README` 文件的历史，使用 `p4v` 的版本图功能。

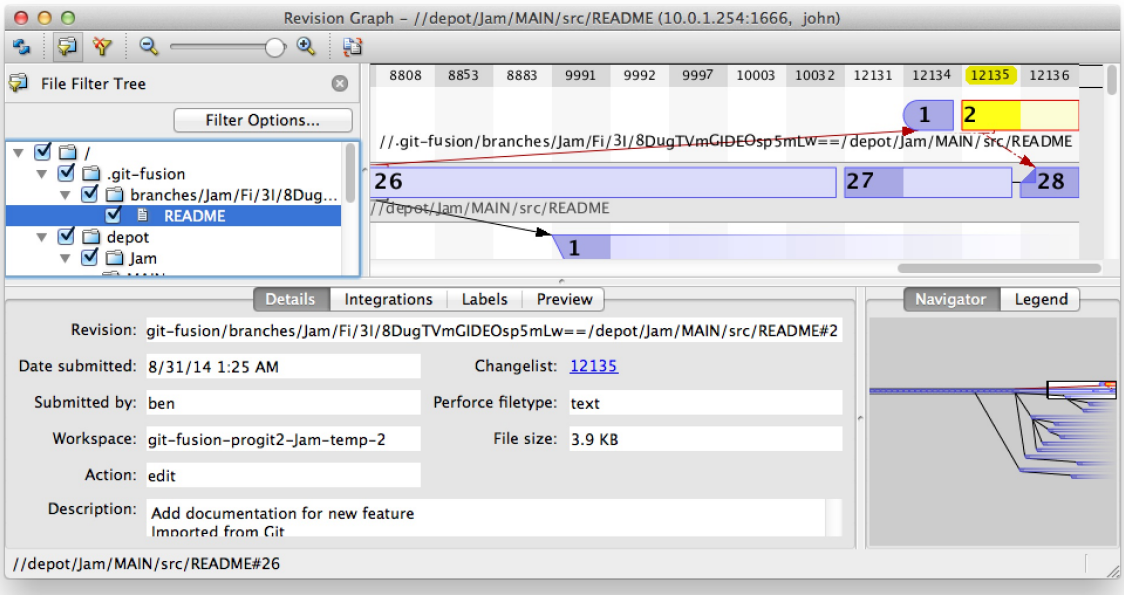


Figure 9-2. Git 推送后的 Perforce 版本图

如果你在之前从未看过这个视图，它似乎让人困惑，但是它显示出了作为 Git 历史图形化查看器相同的概念。 我们正在查看 `README` 文件的历史，所以左上角的目录树只显示那个文件在不同分支的样子。 右上方，我们有不同版本文件关系的可视图，这个可视图的全局视图在右下方。 视图中剩余的部分显示出选择版本的详细信息（在这个例子中是 `2`）

还要注意的一件事是这个图看起来很像 Git 历史中的图。 Perforce 没有存储 `1` 和 `2` 提交的命名分支，所以它在 `.git-fusion` 目录中生成了一个 “anonymous” 分支来保存它。 这也会在 Git 命名分支不对应 Perforce 命名分支时发生（稍后你可以使用配置文件来映射它们到 Perforce 分支）。

这些大多数发生在后台，但是最终结果是团队中的一个人可以使用 Git，另一个可以使用 Perforce，而所有人都不知道其他人的选择。

Git-Fusion 总结

如果你有（或者能获得）接触你的 Perforce 服务器的权限，那么 Git Fusion 是使 Git 与 Perforce 互相交流的很好的方法。 这里包含了一点配置，但是学习曲线并不是很陡峭。 这是本章中其中一个不会出现无法使用 Git 全部能力的警告的章节。 这并不是说扔给 Perforce 任何东西都会高兴 - 如果你尝试重写已经推送的历史，Git Fusion 会拒绝它 - 虽然 Git Fusion 尽力让你感觉是原生的。 你甚至可以使用 Git 子模块（尽管它们对 Perforce 用户看起来很奇怪），合并分支（在 Perforce 这边会被记录了一次整合）。

如果不能说服你的服务器管理员设置 Git Fusion，依然有一种方式来一起使用这两个工具。

Git-p4

Git-p4 是 Git 与 Perforce 之间的双向桥接。 它完全运行在你的 Git 仓库内，所以你不需要任何访问 Perforce 服务器的权限（当然除了用户验证）。 Git-p4 并不像 Git Fusion 一样灵活或完整，但是它允许你在无需修改服务器环境的情况下，做大部分想做的事情。

NOTE

为了与 git-p4 一起工作需要在你的 `PATH` 环境变量中的某个目录中有 `p4` 工具。 在写这篇文章的时候，它可以在 http://www.perforce.com/downloads/Perforce/20-User 免费获得。

设置

出于演示的目的，我们将会从上面演示的 Git Fusion OVA 运行 Perforce 服务器，但是我们会绕过 Git Fusion 服务器然后直接进行 Perforce 版本管理。

为了使用 `p4` 命令行客户端（git-p4 依赖项），你需要设置两个环境变量：

```
$ export P4PORT=10.0.1.254:1666
$ export P4USER=john
```

开始

像在 Git 中的任何事情一样，第一个命令就是克隆：

```
$ git p4 clone //depot/www/live www-shallow
Importing from //depot/www/live into www-shallow
Initialized empty Git repository in /private/tmp/www-shallow/.git/
Doing initial import of //depot/www/live/ from revision #head into refs/remotes/p4/master
```

这样会创建出一种在 Git 中名为 “shallow” 克隆；只有最新版本的 Perforce 被导入至 Git；记住，Perforce 并未被设计成给每一个用户一个版本。 使用 Git 作为 Perforce 客户端这样就足够了，但是为了其他目的的话这样可能不够。

完成之后，我们就有一个全功能的 Git 仓库：

```
$ cd myproject
$ git log --oneline --all --graph --decorate
* 70eaf78 (HEAD, p4/master, p4/HEAD, master) Initial import of //depot/www/live/ from the state at revi
sion #head
```

注意有一个 “p4” 远程代表 Perforce 服务器，但是其他东西看起来就像是标准的克隆。 实际上，这有一点误导；其实远程仓库并不存在。

```
$ git remote -v
```

在当前仓库中并不存在任何远程仓库。 Git-p4 创建了一些引用来代表服务器的状态，它们看起来类似

`git log` 显示的远程引用，但是它们并不被 Git 本身管理，并且你无法推送它们。

工作流程
好了，让我们开始一些工作。 假设你已经在一个非常重要的功能上做了一些工作，然后准备好将它展示给团队中的其他人。

```
$ git log --oneline --all --graph --decorate
* 018467c (HEAD, master) Change page title
* c0fb617 Update link
* 70eaf78 (p4/master, p4/HEAD) Initial import of //depot/www/live/ from the state at revision #head
```

我们已经生成了两次新提交并已准备好推送它们到 Perforce 服务器。 让我们检查一下今天其他人是否做了一些工作：

```
$ git p4 sync
git p4 sync
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12142 (100%)
$ git log --oneline --all --graph --decorate
* 75cd059 (p4/master, p4/HEAD) Update copyright
| * 018467c (HEAD, master) Change page title
| * c0fb617 Update link
|/
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

看起来他们做了，`master` 与 `p4/master` 已经分叉了。 Perforce 的分支系统一点也 不 像 Git 的，所以提交合并提交没有任何意义。 Git-p4 建议变基你的提交，它甚至提供了一个快捷方式来这样做：

```
$ git p4 rebase
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
No changes to import!
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
Applying: Update link
Applying: Change page title
 index.html | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

从输出中可能大概得知，`git p4 rebase` 是 `git p4 sync` 接着 `git rebase p4/master` 的快捷方式。 它比那更聪明一些，特别是工作在多个分支时，但这是一个进步。

现在我们的历史再次是线性的，我们准备好我们的改动贡献回 Perforce。 `git p4 submit` 命令会尝试在 `p4/master` 与 `master` 之间的每一个 Git 提交创建一个新的 Perforce 修订版本。 运行它会带我们到最爱的编辑器，文件内容看起来像是这样：

```
# A Perforce Change Specification.
#
#  Change:      The change number. 'new' on a new changelist.
#  Date:        The date this specification was last modified.
#  Client:      The client on which the changelist was created.  Read-only.
#  User:        The user who created the changelist.
#  Status:      Either 'pending' or 'submitted'. Read-only.
#  Type:        Either 'public' or 'restricted'. Default is 'public'.
#  Description: Comments about the changelist.  Required.
#  Jobs:        What opened jobs are to be closed by this changelist.
#               You may delete jobs from this list.  (New changelists only.)
#  Files:       What opened files from the default changelist are to be added
#               to this changelist.  You may delete files from this list.
#               (New changelists only.)

Change:  new

Client:  john_bens-mbp_8487

User: john

Status:  new

Description:
   Update link

Files:
   //depot/www/live/index.html   # edit


######## git author ben@straub.cc does not match your p4 account.
######## Use option --preserve-user to modify authorship.
######## Variable git-p4.skipUserNameCheck hides this message.
######## everything below this line is just the diff #######
--- //depot/www/live/index.html  2014-08-31 18:26:05.000000000 0000
+++ /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/index.html   2014-08-31 18:26:05.00
0000000 0000
@@ -60,7 +60,7 @@
 </td>
 <td valign=top>
 Source and documentation for
-<a href="http://www.perforce.com/jam/jam.html">
+<a href="jam.html">
 Jam/MR</a>,
 a software build tool.
 </td>
```

除了结尾 git-p4 给我们的帮助性的提示，其它的与你运行 `p4 submit` 后看到的内容大多相同。 当提交或变更集需要一个名字时 git-p4 会分别尝试使用你的 Git 与 Perforce 设置，但是有些情况下你会想要覆盖默认行为。 例如，如果你正导入的提交是由没有 Perforce 用户账户的贡献者编写的，你还是会想要最终的变更集看起来像是他们写的（而不是你）。

Git-p4 帮助性地将 Git 的提交注释导入到 Perforce 变更集的内容，这样所有我们必须做的就是保存并退出，两次（每次一个提交）。 这会使 shell 输出看起来像这样：

```
$ git p4 submit
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-mbp_8487/john_bens-m
bp_8487/depot/www/live/
```

```
Synchronizing p4 checkout...
... - file(s) up-to-date.
Applying dbac45b Update link
//depot/www/live/index.html#4 - opened for edit
Change 12143 created with 1 open file(s).
Submitting change 12143.
Locking 1 files ...
edit //depot/www/live/index.html#5
Change 12143 submitted.
Applying 905ec6a Change page title
//depot/www/live/index.html#5 - opened for edit
Change 12144 created with 1 open file(s).
Submitting change 12144.
Locking 1 files ...
edit //depot/www/live/index.html#6
Change 12144 submitted.
All commits applied!
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12144 (100%)
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
$ git log --oneline --all --graph --decorate
* 775a46f (HEAD, p4/master, p4/HEAD, master) Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

结果恰如我们只是做了一次 `git push`，就像是应当实际发生的最接近的类比。

注意在这个过程中每一个 Git 提交都会被转化为一个 Perforce 变更集；如果想要将它们压缩成为一个单独的提交，可以在运行 `git p4 submit` 前进行一次交互式变基。 同样注意的是所有被转化为变更集的提交的 SHA-1 校验和都改变了；这是因为 git-p4 在每一个转化的提交增加一行到提交注释结尾：

```
$ git log -1
commit 775a46f630d8b46535fc9983cf3ebe6b9aa53145
Author: John Doe <john@example.com>
Date:   Sun Aug 31 10:31:44 2014 -0800

    Change page title

    [git-p4: depot-paths = "//depot/www/live/": change = 12144]
```

当尝试提交一次合并提交时会发生什么？ 让我们尝试一下。 这是我们可能会遇到的一种情形：

```
$ git log --oneline --all --graph --decorate
* 3be6fd8 (HEAD, master) Correct email address
*   1dcbf21 Merge remote-tracking branch 'p4/master'
|\
| * c4689fc (p4/master, p4/HEAD) Grammar fix
* | cbacd0a Table borders: yes please
* | b4959b6 Trademark
|/
* 775a46f Change page title
* 05f1ade Update link
```

```
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Git 与 Perforce 历史在 775a46f 后分叉了。 Git 这边有两次提交，然后一次与 Perforce 头部的合并提交，还有另一个提交。 我们将会尝试提交这些到 Perforce 那边单独的变更集之上。 让我们看看如果尝试这样提交会发生什么：

```
$ git p4 submit -n
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-mbp_8487/john_bens-m
bp_8487/depot/www/live/
Would synchronize p4 checkout in /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/
Would apply
  b4959b6 Trademark
  cbacd0a Table borders: yes please
  3be6fd8 Correct email address
```

`-n` 标记是 `--dry-run` 的缩写，将会报告如果提交命令真的运行会发生什么。 这本例中，它看起来像是我们会创建三个 Perforce 变更集，对应着不在 Perforce 服务器端的三次非合并提交。 那听起来像是我们想要的，让我们看看它会如何做：

```
$ git p4 submit
[…]
$ git log --oneline --all --graph --decorate
* dadbd89 (HEAD, p4/master, p4/HEAD, master) Correct email address
* 1b79a80 Table borders: yes please
* 0097235 Trademark
* c4689fc Grammar fix
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

我们的历史变成线性了，就像在提交前刚刚变基过（实际上也是这样）。 这意味着你可以在 Git 这边自由地创建、工作、扔掉与合并分支而不用害怕你的历史会变得与 Perforce 不兼容。 如果你可以变基它，你就可以将它贡献到 Perforce 服务器。

分支

如果你的 Perforce 项目有多个分支，你并不会不走运；git-p4 可以以一种类似 Git 的方式来处理那种情况。 假定你的 Perforce 仓库平铺的时候像这样：

```
//depot
  └── project
      ├── main
      └── dev
```

并且假定你有一个 `dev` 分支，有一个视图规格像下面这样：

```
//depot/project/main/... //depot/project/dev/...
```

Git-p4 可以自动地检测到这种情形并做正确的事情：

```
$ git p4 clone --detect-branches //depot/project@all
Importing from //depot/project@all into project
Initialized empty Git repository in /private/tmp/project/.git/
Importing revision 20 (50%)
    Importing new branch project/dev

    Resuming with change 20
Importing revision 22 (100%)
Updated branches: main dev
$ cd project; git log --oneline --all --graph --decorate
* eae77ae (HEAD, p4/master, p4/HEAD, master) main
| * 10d55fb (p4/project/dev) dev
| * a43cfae Populate //depot/project/main/... //depot/project/dev/....
|/
* 2b83451 Project init
```

注意在仓库路径中的 “@all” 说明符；那会告诉 git-p4 不仅仅只是克隆那个子树最新的变更集，更包括那些路径未接触的所有变更集。 这有点类似于 Git 的克隆概念，但是如果你工作在一个具有很长历史的项目，那么它会花费一段时间。

`--detect-branches` 标记告诉 git-p4 使用 Perforce 的分支规范来映射到 Git 的引用中。 如果这些映射不在 Perforce 服务器中（使用 Perforce 的一种完美有效的方式），你可以告诉 git-p4 分支映射是什么，然后你会得到同样的结果：

```
$ git init project
Initialized empty Git repository in /tmp/project/.git/
$ cd project
$ git config git-p4.branchList main:dev
$ git clone --detect-branches //depot/project@all .
```

设置 `git-p4.branchList` 配置选项为 `main:dev` 告诉 git-p4 那个 “main” 与 “dev” 都是分支，第二个是第一个的子分支。

如果我们现在运行 `git checkout -b dev p4/project/dev` 并且做一些提交，在运行 `git p4 submit` 时 git-p4 会聪明地选择正确的分支。 不幸的是，git-p4 不能混用 shallow 克隆与多个分支；如果你有一个巨型项目并且想要同时工作在不止一个分支上，可能不得不针对每一个你想要提交的分支运行一次 `git p4 clone`。

为了创建与整合分支，你不得不使用一个 Perforce 客户端。 Git-p4 只能同步或提交已有分支，并且它一次只能做一个线性的变更集。 如果你在 Git 中合并两个分支并尝试提交新的变更集，所有这些会被记录为一串文件修改；关于哪个分支参与的元数据在整合中会丢失。

Git 与 Perforce 总结

Git-p4 将与 Perforce 服务器工作时使用 Git 工作流成为可能，并且它非常擅长这点。 然而，需要记住的重要一点是 Perforce 负责源头，而你只是在本地使用 Git。 在共享 Git 提交时要相当小心：如果你有一个其他人使用的远程仓库，不要在提交到 Perforce 服务器前推送任何提交。

如果想要为源码管理自由地混合使用 Perforce 与 Git 作为客户端，可以说服服务器管理员安装 Git Fusion，Git Fusion 使 Git 作为 Perforce 服务器的首级版本管理客户端。

Git 与 TFS

Git 在 Windows 开发者当中变得流行起来，如果你正在 Windows 上编写代码并且正在使用 Microsoft 的 Team Foundation Server (TFS)，这会是个好机会。 TFS 是一个包含工作项目检测与跟踪、支持 Scrum 与其他流程管理方法、代码审核、版本控制的协作套件。 这里有一点困惑：TFS是服务器，它支持通过 Git 与它们自定义的 VCS 来管理源代码，这被他们称为 TFVC（Team Foundation Version Control）。 Git 支持 TFS（自 2013 版本起）的部分新功能，所以在那之前所有工具都将版本控制部分称为 “TFS” ，即使实际上他们大部分时间都在与 TFVC 工作。

如果发现你的团队在使用 TFVC 但是你更愿意使用 Git 作为版本控制客户端，这里为你准备了一个项目。

选择哪个工具
实际上，这里有两个工具：git-tf 与 git-tfs。

Git-tfs （可以在 https://github.com/git-tfs/git-tfs 找到）是一个 .NET 项目，它只能运行在 Windows 上（截至文章完成时）。 为了操作 Git 仓库，它使用了 libgit2 的 .NET 绑定，一个可靠的面向库的 Git 实现，十分灵活且性能优越。 Libgit2 并不是一个完整的 Git 实现，为了弥补差距 git-tfs 实际上会调用 Git 命令行客户端来执行某些操作，因此在操作 Git 仓库时并没有任何功能限制。 因为它使用 Visual Studio 程序集对服务器进行操作，所以它对 TFVC 的支持非常成熟。 这并不意味着你需要接触那些程序集，但是意味着你需要安装 Visual Studio 的一个最近版本（2010 之后的任何版本，包括 2012 之后的 Express 版本），或者 Visual Studio SDK。

Git-tf（主页在 https://gittf.codeplex.com）是一个 Java 项目，因此它可以运行在任何一个有 Java 运行时环境的电脑上。 它通过 JGit（一个 Git 的 JVM 实现）来与 Git 仓库交互，这意味着事实上它没有 Git 功能上的限制。 然而，相对于 git-tfs 它对 TFVC 的支持是有限的 - 例如，它不支持分支。

所以每个工具都有优点和缺点，每个工具都有它适用的情况。 我们在本书中将会介绍它们两个的基本用法。

NOTE

你需要有一个基于 TFVC 的仓库来执行后续的指令。 现实中它们并没有 Git 或 Subversion 仓库那样多，所以你可能需要创建一个你自己的仓库。 Codeplex (https://www.codeplex.com) 或 Visual Studio Online (http://www.visualstudio.com) 都是非常好的选择。

使用：`git-tf`
和其它任何 Git 项目一样，你要做的第一件事是克隆。 使用 `git-tf` 克隆看起来像这样：

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main project_git
```

第一个参数是一个 TFVC 集的 URL，第二个参数类似于 `$/project/branch` 的形式，第三个参数是将要创建的本地 Git 仓库路径（最后一项可以省略）。 Git-tf 同一时间只能工作在一个分支上；如果你想要检入一个不同的 TFVC 分支，你需要从那个分支克隆一份新的。

这会创建一个完整功能的 Git 仓库：

```
$ cd project_git
$ git log --all --oneline --decorate
512e75a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Checkin message
```

这叫做 浅 克隆，意味着只下载了最新的变更集。 TFVC 并未设计成为每一个客户端提供一份全部历史记录的拷贝，所以 git-tf 默认行为是获得最新的版本，这样更快一些。

如果愿意多花一些时间，使用 `--deep` 选项克隆整个项目历史可能更有价值。

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main \
  project_git --deep
Username: domain\user
Password:
Connecting to TFS...
Cloning $/myproject into /tmp/project_git: 100%, done.
Cloned 4 changesets. Cloned last changeset 35190 as d44b17a
$ cd project_git
$ git log --all --oneline --decorate
d44b17a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Goodbye
126aa7b (tag: TFS_C35189)
8f77431 (tag: TFS_C35178) FIRST
0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
        Team Project Creation Wizard
```

注意名字类似 `TFS_C35189` 的标签；这是一个帮助你知道 Git 提交与 TFVC 变更集关联的功能。 这是一种优雅的表示方式，因为通过一个简单的 log 命令就可以看到你的提交是如何与 TFVC 中已存在快照关联起来的。 它们并不是必须的（并且实际上可以使用 `git config git-tf.tag false` 来关闭它们）- git-tf 会在 `.git/git-tf` 文件中保存真正的提交与变更集的映射。

使用：**`git-tfs`**
Git-tfs 克隆行为略为不同。 观察：

```
PS> git tfs clone --with-branches \
    https://username.visualstudio.com/DefaultCollection \
    $/project/Trunk project_git
Initialized empty Git repository in C:/Users/ben/project_git/.git/
C15 = b75da1aba1ffb359d00e85c52acb261e4586b0c9
C16 = c403405f4989d73a2c3c119e79021cb2104ce44a
Tfs branches found:
```

```
- $/tfvc-test/featureA
The name of the local branch will be : featureA
C17 = d202b53f67bde32171d5078968c644e562f1c439
C18 = 44cd729d8df868a8be20438fdeeefb961958b674
```

注意 `--with-branches` 选项。 Git-tfs 能够映射 TFVC 分支到 Git 分支，这个标记告诉它为每一个 TFVC 分支建立一个本地的 Git 分支。 强烈推荐曾经在 TFS 中新建过分支或合并过分支的仓库使用这个标记，但是如果使用的服务器的版本比 TFS 2010 更老 - 在那个版本前，“分支” 只是文件夹，所以 git-tfs 无法将它们与普通文件夹区分开。

让我们看一下最终的 Git 仓库：

```
PS> git log --oneline --graph --decorate --all
* 44cd729 (tfs/featureA, featureA) Goodbye
* d202b53 Branched from $/tfvc-test/Trunk
* c403405 (HEAD, tfs/default, master) Hello
* b75da1a New project
PS> git log -1
commit c403405f4989d73a2c3c119e79021cb2104ce44a
Author: Ben Straub <ben@straub.cc>
Date:   Fri Aug 1 03:41:59 2014 +0000

    Hello

    git-tfs-id: [https://username.visualstudio.com/DefaultCollection]$/myproject/Trunk;C16
```

有两个本地分支，`master` 与 `featureA`，分别代表着克隆（TFVC 中的 `Trunk`）与子分支（TFVC 中的 `featureA`）的初始状态。 也可以看到 `tfs` “remote” 也有一对引用：`default` 与 `featureA`，代表 TFVC 分支。 Git-tfs 映射从 `tfs/default` 克隆的分支，其他的会有它们自己的名字。

另一件需要注意的事情是在提交信息中的 `git-tfs-id:` 行。 Git-tfs 使用这些标记而不是标签来关联 TFVC 变更集与 Git 提交。 有一个潜在的问题是 Git 提交在推送到 TFVC 前后会有不同的 SHA-1 校验和。

Git-tf[s] 工作流程

NOTE

无论你使用哪个工具，都需要先设置几个 Git 配置选项来避免一些问题。

```
$ git config set --local core.ignorecase=true
$ git config set --local core.autocrlf=false
```

显然，接下来要做的事情就是要在项目中做一些工作。 TFVC 与 TFS 有几个功能可能会增加你的工作流程的复杂性：

1. TFVC 无法表示特性分支，这会增加一点复杂度。 这会导致需要以 非常 不同的方式使用 TFVC 与 Git

表示的分支。

2. 要意识到 TFVC 允许用户从服务器上 “检出” 文件并锁定它们，这样其他人就无法编辑了。 显然它不会阻止你在本地仓库中编辑它们，但是当推送你的修改到 TFVC 服务器时会出现问题。

3. TFS 有一个 “封闭” 检入的概念，TFS 构建-测试循环必须在检入被允许前成功完成。 这使用了 TFVC 的 “shelve” 功能，我们不会在这里详述。 可以通过 git-tf 手动地模拟这个功能，并且 git-tfs 提供了封闭敏感的 `checkintool` 命令。

出于简洁性的原因，我们这里介绍的是一种轻松的方式，回避并避免了大部分问题。

工作流程：**`git-tf`**
假定你完成了一些工作，在 `master` 中做了几次 Git 提交，然后准备将你的进度共享到服务器。 这是我们的 Git 仓库：

```
$ git log --oneline --graph --decorate --all
* 4178a82 (HEAD, master) update code
* 9df2ae3 update readme
* d44b17a (tag: TFS_C35190, origin_tfs/tfs) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
          Team Project Creation Wizard
```

我们想要拿到在 `4178a82` 提交的快照并将其推送到 TFVC 服务器。 先说重要的：让我们看看自从上次连接后我们的队友是否进行过改动：

```
$ git tf fetch
Username: domain\user
Password:
Connecting to TFS...
Fetching $/myproject at latest changeset: 100%, done.
Downloaded changeset 35320 as commit 8ef06a8. Updated FETCH_HEAD.
$ git log --oneline --graph --decorate --all
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
| * 4178a82 (HEAD, master) update code
| * 9df2ae3 update readme
|/
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
          Team Project Creation Wizard
```

看起来其他人也做了一些改动，现在我们有一个分叉的历史。 这就是 Git 的优势，但是我们现在有两种处理的方式：

1. 像一名 Git 用户一样自然的生成一个合并提交（毕竟，那也是 `git pull` 做的），git-tf 可以通过一个简单的 `git tf pull` 来帮你完成。 然而，我们要注意的是，TFVC 却并不这样想，如果你推

送合并提交那么你的历史在两边看起来都不一样，这会造成困惑。 其次，如果你计划将所有你的改动提交为一次变更集，这可能是最简单的选择。

2. 变基使我们的提交历史变成直线，这意味着我们有个选项可以将我们的每一个 Git 提交转换为一个 TFVC 变更集。 因为这种方式为其他选项留下了可能，所以我们推荐你这样做；git-tf 可以很简单地通过 `git tf pull --rebase` 帮你达成目标。

这是你的选择。 在本例中，我们会进行变基：

```
$ git rebase FETCH_HEAD
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
          Team Project Creation Wizard
```

现在我们准备好生成一个检入来推送到 TFVC 服务器上了。 Git-tf 给你一个将自上次修改（即 `--shallow` 选项，默认启用）以来所有的修改生成的一个单独的变更集以及为每一个 Git 提交（`--deep`）生成的一个新的变更集。 在本例中，我们将会创建一个变更集：

```
$ git tf checkin -m 'Updating readme and code'
Username: domain\user
Password:
Connecting to TFS...
Checking in to $/myproject: 100%, done.
Checked commit 5a0e25e in as changeset 35348
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, tag: TFS_C35348, origin_tfs/tfs, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
          Team Project Creation Wizard
```

那有一个新标签 `TFS_C35348`，表明 TFVC 已经存储了一个相当于 `5a0e25e` 提交的快照。 要重点注意的是，不是每一个 Git 提交都需要在 TFVC 中存在一个相同的副本；例如 `6eb3eb5` 提交，在服务器上并不存在。

这就是主要的工作流程。 有一些你需要考虑的其他注意事项：

- 没有分支。 Git-tf 同一时间只能从一个 TFVC 分支创建一个 Git 仓库。

- 协作时使用 TFVC 或 Git，而不是两者同时使用。 同一个 TFVC 仓库的不同 git-tf 克隆会有不同的 SHA-1 校验和，这会导致无尽的头痛问题。
- 如果你的团队的工作流程包括在 Git 中协作并定期与 TFVC 同步，只能使用其中的一个 Git 仓库连接到 TFVC。

工作流程：**`git-tfs`**
让我们使用 git-tfs 来走一遍同样的情景。 这是我们在 Git 仓库中 `master` 分支上生成的几个新提交：

```
PS> git log --oneline --graph --all --decorate
* c3bd3ae (HEAD, master) update code
* d85e5a2 update readme
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 (tfs/default) Hello
* b75da1a New project
```

让我们看一下在我们工作时有没有人完成一些其它的工作：

```
PS> git tfs fetch
C19 = aea74a0313de0a391940c999e51c5c15c381d91d
PS> git log --all --oneline --graph --decorate
* aea74a0 (tfs/default) update documentation
| * c3bd3ae (HEAD, master) update code
| * d85e5a2 update readme
|/
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

是的，那说明我们的同事增加了一个新的 TFVC 变更集，显示为新的 `aea74a0` 提交，而 `tfs/default` 远程分支已经被移除了。

与 git-tf 相同，我们有两种基础选项来解决这个分叉历史问题：

1. 通过变基来保持历史是线性的。
2. 通过合并来保留改动。

在本例中，我们将要做一个 “深” 检入，也就是说每一个 Git 提交会变成一个 TFVC 变更集，所以我们想要变基。

```
PS> git rebase tfs/default
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
```

```
PS> git log --all --oneline --graph --decorate
* 10a75ac (HEAD, master) update code
* 5cec4ab update readme
* aea74a0 (tfs/default) update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

现在已经准备好通过检入我们的代码到 TFVC 服务器来完成贡献。 我们这里将会使用 `rcheckin` 命令将 HEAD 到第一个 `tfs` 远程分支间的每一个 Git 提交转换为一个 TFVC 变更集（ `checkin` 命令只会创建一个变更集，有些类似于压缩 Git 提交）。

```
PS> git tfs rcheckin
Working with tfs remote: default
Fetching changes from TFS to minimize possibility of late conflict...
Starting checkin of 5cec4ab4 'update readme'
 add README.md
C20 = 71a5ddce274c19f8fdc322b4f165d93d89121017
Done with 5cec4ab4b213c354341f66c80cd650ab98dcf1ed, rebasing tail onto new TFS-commit...
Rebase done successfully.
Starting checkin of b1bf0f99 'update code'
 edit .git\tfs\default\workspace\ConsoleApplication1/ConsoleApplication1/Program.cs
C21 = ff04e7c35dfbe6a8f94e782bf5e0031cee8d103b
Done with b1bf0f9977b2d48bad611ed4a03d3738df05ea5d, rebasing tail onto new TFS-commit...
Rebase done successfully.
No more to rcheckin.
PS> git log --all --oneline --graph --decorate
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

注意在每次成功检入到 TFVC 服务器后，git-tfs 是如何将剩余的工作变基到服务器上。 这是因为它将 `git-tfs-id` 属性加入到提交信息的底部，这将会改变 SHA-1 校验和。 这恰恰是有意设计的，没有什么事情可以担心了，但是你应该意识到发生了什么，特别是当你想要与其他人共享 Git 提交时。

TFS 有许多与它的版本管理系统整合的功能，比如工作项目、指定审核者、封闭检入等等。 仅仅通过命令行工具使用这些功能来工作是很笨重的，但是幸运的是 git-tfs 允许你轻松地运行一个图形化的检入工具：

```
PS> git tfs checkintool
PS> git tfs ct
```

它看起来有点像这样：

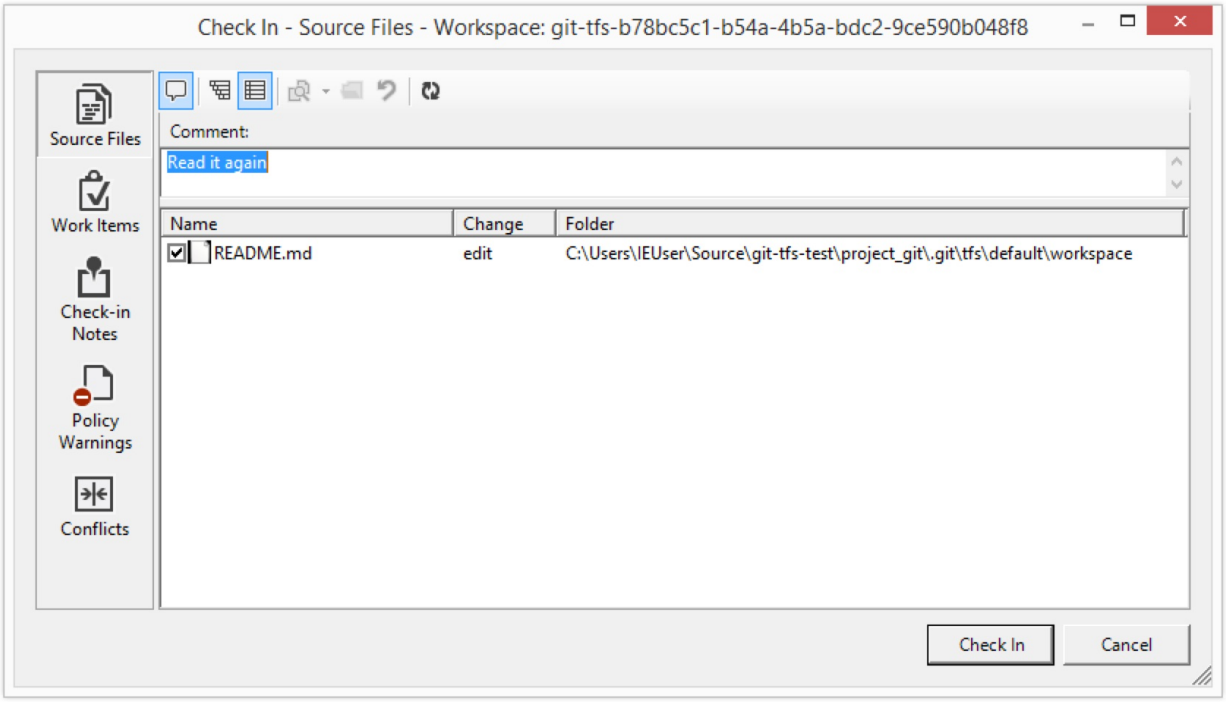


Figure 9-3. git-tfs 检入工具。

对 TFS 用户来说这看起来很熟悉，因为它就是从 Visual Studio 中运行的同一个窗口。

Git-tfs 同样允许你从你的 Git 仓库控制 TFVC 分支。 如同这个例子，让我们创建一个：

```
PS> git tfs branch $/tfvc-test/featureBee
The name of the local branch will be : featureBee
C26 = 1d54865c397608c004a2cadce7296f5edc22a7e5
PS> git log --oneline --graph --decorate --all
* 1d54865 (tfs/featureBee) Creation branch $/myproject/featureBee
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

在 TFVC 中创建一个分支意味着增加一个使分支存在的变更集，这会映射为一个 Git 提交。 也要注意的是 git-tfs 创建 了 `tfs/featureBee` 远程分支，但是 `HEAD` 始终指向 `master`。 如果你想要在新生成的分支上工作，那你也许应该通过从那次提交创建一个特性分支的方式使你新的提交基于 `1d54865` 提交。

Git 与 TFS 总结

Git-tf 与 Git-tfs 都是与 TFVC 服务器交互的很好的工具。 它们允许你在本地使用 Git 的能力，避免与中

央 TFVC 服务器频繁交流，使你做为一个开发者的生活更轻松，而不用强制整个团队迁移到 Git。 如果你在 Windows 上工作（那很有可能你的团队正在使用 TFS），你可能会想要使用 git-tfs，因为它的功能更完整，但是如果你在其他平台工作，你只能使用略有限制的 git-tf。 像本章中大多数工具一样，你应当使用其中的一个版本系统作为主要的，而使用另一个做为次要的 - 不管是 Git 还是 TFVC 都可以做为协作中心，但不是两者都用。

9.2 迁移到 Git

迁移到 Git

如果你现在有一个正在使用其他 VCS 的代码库，但是你已经决定开始使用 Git，必须通过某种方式将你的项目迁移至 Git。 这一部分会介绍一些通用系统的导入器，然后演示如何开发你自己定制的导入器。 你将会学习如何从几个大型专业应用的 SCM 系统中导入数据，不仅因为它们是大多数想要转换的用户正在使用的系统，也因为获取针对它们的高质量工具很容易。

Subversion

如果你阅读过前面关于 `git svn` 的章节，可以轻松地使用那些指令来 `git svn clone` 一个仓库，停止使用 Subversion 服务器，推送到一个新的 Git 服务器，然后就可以开始使用了。 如果你想要历史，可以从 Subversion 服务器上尽可能快地拉取数据来完成这件事（这可能会花费一些时间）。

然而，导入并不完美；因为花费太长时间了，你可能早已用其他方法完成导入操作。 导入产生的第一个问题就是作者信息。 在 Subversion 中，每一个人提交时都需要在系统中有一个用户，它会被记录在提交信息内。 在之前章节的例子中几个地方显示了 `schacon`，比如 `blame` 输出与 `git svn log`。 如果想要将上面的 Subversion 用户映射到一个更好的 Git 作者数据中，你需要一个 Subversion 用户到 Git 用户的映射。 创建一个 `users.txt` 的文件包含像下面这种格式的映射：

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

为了获得 SVN 使用的作者名字列表，可以运行这个：

```
$ svn log --xml | grep author | sort -u | \
  perl -pe 's/.*>(.*?)<.*/$1 = /'
```

这会将日志输出为 XML 格式，然后保留作者信息行、去除重复、去除 XML 标记。 （很显然这只会在安装了 `grep`、 `sort` 与 `perl` 的机器上运行。）然后，将输出重定向到你的 users.txt 文件中，这样就可以在每一个记录后面加入对应的 Git 用户数据。

你可以将此文件提供给 `git svn` 来帮助它更加精确地映射作者数据。 也可以通过传递 `--no-metadata` 给 `clone` 与 `init` 命令，告诉 `git svn` 不要包括 Subversion 通常会导入的元数据。 这会使你的 `import` 命令看起来像这样：

```
$ git svn clone http://my-project.googlecode.com/svn/ \
      --authors-file=users.txt --no-metadata -s my_project
```

现在在 `my_project` 目录中应当有了一个更好的 Subversion 导入。 并不像是下面这样的提交：

```
commit 37efa680e8473b615de980fa935944215428a35a
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk

    git-svn-id: https://my-project.googlecode.com/svn/trunk@94 4c93b258-373f-11de-
    be05-5f7a86268029
```

反而它们看起来像是这样：

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk
```

不仅是 Author 字段更好看了， `git-svn-id` 也不在了。

之后，你应当做一些导入后的清理工作。 第一步，你应当清理 `git svn` 设置的奇怪的引用。 首先移动标签，这样它们就是标签而不是奇怪的远程引用，然后你会移动剩余的分支这样它们就是本地的了。

为了将标签变为合适的 Git 标签，运行

```
$ cp -Rf .git/refs/remotes/origin/tags/* .git/refs/tags/
$ rm -Rf .git/refs/remotes/origin/tags
```

这会使原来在 `remotes/origin/tags/` 里的远程分支引用变成真正的（轻量）标签。

接下来，将 `refs/remotes` 下剩余的引用移动为本地分支：

```
$ cp -Rf .git/refs/remotes/* .git/refs/heads/
$ rm -Rf .git/refs/remotes
```

现在所有的旧分支都是真正的 Git 分支，并且所有的旧标签都是真正的 Git 标签。 最后一件要做的事情是，将你的新 Git 服务器添加为远程仓库并推送到上面。 下面是一个将你的服务器添加为远程仓库的例子：

```
$ git remote add origin git@my-git-server:myrepository.git
```

因为想要上传所有分支与标签，你现在可以运行：

```
$ git push origin --all
```

通过以上漂亮、干净地导入操作，你的所有分支与标签都应该在新 Git 服务器上。

Mercurial

因为 Mercurial 与 Git 在表示版本时有着非常相似的模型，也因为 Git 拥有更加强大的灵活性，将一个仓库从 Mercurial 转换到 Git 是相当直接的，使用一个叫作“hg-fast-export”的工具，需要从这里拷贝一份：

```
$ git clone http://repo.or.cz/r/fast-export.git /tmp/fast-export
```

转换的第一步就是要先得到想要转换的 Mercurial 仓库的完整克隆：

```
$ hg clone <remote repo URL> /tmp/hg-repo
```

下一步就是创建一个作者映射文件。 Mercurial 对放入到变更集作者字段的内容比 Git 更宽容一些，所以这是一个清理的好机会。 只需要用到 `bash` 终端下的一行命令：

```
$ cd /tmp/hg-repo
$ hg log | grep user: | sort | uniq | sed 's/user: *//' > ../authors
```

这会花费几秒钟，具体要看项目提交历史有多少，最终 `/tmp/authors` 文件看起来会像这样：

```
bob
bob@localhost
bob <bob@company.com>
bob jones <bob <AT> company <DOT> com>
Bob Jones <bob@company.com>
Joe Smith <joe@company.com>
```

在这个例子中，同一个人（Bob）使用不同的名字创建变更集，其中一个实际上是正确的，另一个完全不符合 Git 提交的规范。 Hg-fast-export 通过向我们想要修改的行尾添加 `={new name and email address}` 来修正这个问题，移除任何我们想要保留的用户名所在的行。 如果所有的用户名看起来都是正确的，那我们根本就不需要这个文件。 在本例中，我们会使文件看起来像这样：

```
bob=Bob Jones <bob@company.com>
bob@localhost=Bob Jones <bob@company.com>
bob jones <bob <AT> company <DOT> com>=Bob Jones <bob@company.com>
bob <bob@company.com>=Bob Jones <bob@company.com>
```

下一步是创建一个新的 Git 仓库，然后运行导出脚本：

```
$ git init /tmp/converted
$ cd /tmp/converted
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
```

`-r` 选项告诉 hg-fast-export 去哪里寻找我们想要转换的 Mercurial 仓库，`-A` 标记告诉它在哪找到作者映射文件。 这个脚本会分析 Mercurial 变更集然后将它们转换成 Git“fast-import”功能（我们将在之后详细讨论）需要的脚本。 这会花一点时间（尽管它比通过网格 更 快），输出相当的冗长：

```
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
Loaded 4 authors
master: Exporting full revision 1/22208 with 13/0/0 added/changed/removed files
master: Exporting simple delta revision 2/22208 with 1/1/0 added/changed/removed files
master: Exporting simple delta revision 3/22208 with 0/1/0 added/changed/removed files
[…]
master: Exporting simple delta revision 22206/22208 with 0/4/0 added/changed/removed files
master: Exporting simple delta revision 22207/22208 with 0/2/0 added/changed/removed files
master: Exporting thorough delta revision 22208/22208 with 3/213/0 added/changed/removed files
Exporting tag [0.4c] at [hg r9] [git :10]
Exporting tag [0.4d] at [hg r16] [git :17]
[…]
Exporting tag [3.1-rc] at [hg r21926] [git :21927]
Exporting tag [3.1] at [hg r21973] [git :21974]
Issued 22315 commands
git-fast-import statistics:
---------------------------------------------------------------------
Alloc'd objects:     120000
Total objects:       115032 (    208171 duplicates                 )
      blobs  :        40504 (    205320 duplicates      26117 deltas of      39602 attempts)
      trees  :        52320 (      2851 duplicates      47467 deltas of      47599 attempts)
      commits:        22208 (         0 duplicates          0 deltas of          0 attempts)
      tags   :            0 (         0 duplicates          0 deltas of          0 attempts)
Total branches:         109 (         2 loads     )
      marks:        1048576 (     22208 unique    )
      atoms:           1952
Memory total:          7860 KiB
       pools:          2235 KiB
     objects:          5625 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit      = 8589934592
pack_report: pack_used_ctr            =      90430
pack_report: pack_mmap_calls          =      46771
pack_report: pack_open_windows        =          1 /          1
pack_report: pack_mapped              =  340852700 /  340852700
---------------------------------------------------------------------

$ git shortlog -sn
   369  Bob Jones
   365  Joe Smith
```

那看起来非常好。 所有 Mercurial 标签都已被转换成 Git 标签，Mercurial 分支与书签都被转换成 Git 分支。 现在已经准备好将仓库推送到新的服务器那边：

```
$ git remote add origin git@my-git-server:myrepository.git
$ git push origin --all
```

Perforce

下一个将要看到导入的系统是 Perforce。 就像我们之前讨论过的，有两种方式让 Git 与 Perforce 互相通信：git-p4 与 Perforce Git Fusion。

Perforce Git Fusion

Git Fusion 使这个过程毫无痛苦。 只需要使用在 Git Fusion 中讨论过的配置文件来配置你的项目设置、用户映射与分支，然后克隆整个仓库。 Git Fusion 让你处在一个看起来像是原生 Git 仓库的环境中，如果愿意的话你可以随时将它推送到一个原生 Git 托管中。 如果你喜欢的话甚至可以使用 Perforce 作为你的 Git 托管。

Git-p4

Git-p4 也可以作为一个导入工具。 作为例子，我们将从 Perforce 公开仓库中导入 Jam 项目。 为了设置客户端，必须导出 P4PORT 环境变量指向 Perforce 仓库：

```
$ export P4PORT=public.perforce.com:1666
```

NOTE

为了继续后续步骤，需要连接到 Perforce 仓库。 在我们的例子中将会使用在 public.perforce.com 的公开仓库，但是你可以使用任何你有权限的仓库。

运行 `git p4 clone` 命令从 Perforce 服务器导入 Jam 项目，提供仓库、项目路径与你想要存放导入项目的路径：

```
$ git-p4 clone //guest/perforce_software/jam@all p4import
Importing from //guest/perforce_software/jam@all into p4import
Initialized empty Git repository in /private/tmp/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 9957 (100%)
```

这个特定的项目只有一个分支，但是如果你在分支视图（或者说一些目录）中配置了一些分支，你可以将 `--detect-branches` 选项传递给 `git p4 clone` 来导入项目的所有分支。 查看 分支 来了解关于这点的更多信息。

此时你几乎已经完成了。 如果进入 `p4import` 目录中并运行 `git log`，可以看到你的导入工作：

```
$ git log -2
commit e5da1c909e5db3036475419f6379f2c73710c4e6
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800
```

```
    Correction to line 355; change </UL> to </OL>.

    [git-p4: depot-paths = "//public/jam/src/": change = 8068]

commit aa21359a0a135dda85c50a7f7cf249e4f7b8fd98
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800

    Fix spelling error on Jam doc page (cummulative -> cumulative).

    [git-p4: depot-paths = "//public/jam/src/": change = 7304]
```

你可以看到 `git-p4` 在每一个提交里都留下了一个标识符。 如果之后想要引用 Perforce 的修改序号的话，标识符保留在那里也是可以的。 然而，如果想要移除标识符，现在正是这么做的时候 - 在你开始在新仓库中工作之前。 可以使用 `git filter-branch` 将全部标识符移除。

```
$ git filter-branch --msg-filter 'sed -e "/^\[git-p4:/d"'
Rewrite e5da1c909e5db3036475419f6379f2c73710c4e6 (125/125)
Ref 'refs/heads/master' was rewritten
```

如果运行 `git log`，你会看到所有提交的 SHA-1 校验和都改变了，但是提交信息中不再有 `git-p4` 字符串了：

```
$ git log -2
commit b17341801ed838d97f7800a54a6f9b95750839b7
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800

    Correction to line 355; change </UL> to </OL>.

commit 3e68c2e26cd89cb983eb52c024ecdfba1d6b3fff
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800

    Fix spelling error on Jam doc page (cummulative -> cumulative).
```

现在导入已经准备好推送到你的新 Git 服务器上了。

TFS

如果你的团队正在将他们的源代码管理从 TFVC 转换为 Git，你们会想要最高程度的无损转换。 这意味着，虽然我们在之前的交互章节介绍了 git-tfs 与 git-tf 两种工具，但是我们在本部分只能介绍 git-tfs，因为 git-tfs 支持分支，而使用 git-tf 代价太大。

NOTE

这是一个单向转换。 这意味着 Git 仓库无法连接到原始的 TFVC 项目。

第一件事是映射用户名。 TFVC 对待变更集作者字段的内容相当宽容，但是 Git 需要人类可读的名字与邮箱地址。 可以通过 `tf` 命令行客户端来获取这个信息，像这样：

```
PS> tf history $/myproject -recursive > AUTHORS_TMP
```

这会将历史中的所有变更集抓取下来并放到 AUTHORS_TMP 文件中，然后我们将会将 `User` 列（第二个）取出来。 打开文件找到列开始与结束的字符并替换，在下面的命令行中，`cut` 命令的参数 `11-20` 就是我们找到的：

```
PS> cat AUTHORS_TMP | cut -b 11-20 | tail -n+3 | uniq | sort > AUTHORS
```

`cut` 命令只会保留每行中第 11 个到第 22 个字符。 `tail` 命令会跳过前两行，就是字段表头与 ASCII 风格的下划线。 所有这些的结果通过管道送到 `uniq` 来去除重复，然后保存到 `AUTOHRS` 文件中。 下一步是手动的；为了让 git-tfs 有效地使用这个文件，每一行必须是这种格式：

```
DOMAIN\username = User Name <email@address.com>
```

左边的部分是 TFVC 中的 “User” 字段，等号右边的部分是将被用作 Git 提交的用户名。

一旦有了这个文件，下一件事就是生成一个你需要的 TFVC 项目的完整克隆：

```
PS> git tfs clone --with-branches --authors=AUTHORS https://username.visualstudio.com/DefaultCollection
 $/project/Trunk project_git
```

接下来要从提交信息底部清理 `git-tfs-id` 区块。 下面的命令会完成这个任务：

```
PS> git filter-branch -f --msg-filter 'sed "s/^git-tfs-id:.*$//g"' -- --all
```

那会使用 Git 终端环境中的 `sed` 命令来将所有以 “git-tfs-id:” 开头的行替换为 Git 会忽略的空白。

全部完成后，你就已经准备好去增加一个新的远程仓库，推送你所有的分支上去，然后你的团队就可以开始用 Git 工作了。

一个自定义的导入器

如果你的系统不是上述中的任何一个，你需要在线查找一个导入器 - 针对许多其他系统有很多高质量的导入器，包括 CVS、Clear Case、Visual Source Safe，甚至是一个档案目录。 如果没有一个工具适合你，需要一个不知名的工具，或者需要更大自由度的自定义导入过程，应当使用 `git fast-import`。 这个命令从标准输入中读取简单指令来写入特定的 Git 数据。 通过这种方式创建 Git 对象比运行原始 Git 命令或直接写入原始对象（查看 Git 内部原理 了解更多内容）更容易些。 通过这种方式你可以编写导入脚本，从你要导入的系统中读取必要数据，然后直接打印指令到标准输出。 然后可以运行这个程序并通过 `git fast-import` 重定向管道输出。

为了快速演示，我们会写一个简单的导入器。 假设你在 `current` 工作，有时候会备份你的项目到时间标签 `back_YYYY_MM_DD` 备份目录中，你想要将这些导入到 Git 中。 目录结构看起来是这样：

```
$ ls /opt/import_from
back_2014_01_02
back_2014_01_04
back_2014_01_14
back_2014_02_03
current
```

为了导入一个 Git 目录，需要了解 Git 如何存储它的数据。 你可能记得，Git 在底层存储指向内容快照的提交对象的链表。 所有要做的就是告诉 `fast-import` 哪些内容是快照，哪个提交数据指向它们，以及它们进入的顺序。 你的策略是一次访问一个快照，然后用每个目录中的内容创建提交，并且将每一个提交与前一个连接起来。

如同我们在 使用强制策略的一个例子 里做的，我们将会使用 Ruby 写这个，因为它是我们平常工作中使用的并且它很容易读懂。 可以使用任何你熟悉的东西来非常轻松地写这个例子 - 它只需要将合适的信息打印到 `标准输出` 。 然而，如果你在 Windows 上，这意味着需要特别注意不要引入回车符到行尾 - git fast-import 非常特别地只接受换行符（LF）而不是 Windows 使用的回车换行符（CRLF）。

现在开始，需要进入目标目录中并识别每一个子目录，每一个都是你要导入为提交的快照。 要进入到每个子目录中并为导出它打印必要的命令。 基本主循环像这个样子：

```
last_mark = nil

# loop through the directories
Dir.chdir(ARGV[0]) do
  Dir.glob("*").each do |dir|
    next if File.file?(dir)

    # move into the target directory
    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end
```

在每个目录内运行 `print_export` ，将会拿到清单并标记之前的快照，然后返回清单并标记现在的快照；通过这种方式，可以将它们合适地连接在一起。 “标记” 是一个给提交标识符的 `fast-import` 术语；当你创建提交，为每一个提交赋予一个标记来将它与其他提交连接在一起。 这样，在你的 `print_export` 方法中第一件要做的事就是从目录名字生成一个标记：

```
mark = convert_dir_to_mark(dir)
```

可以创建一个目录的数组并使用索引做为标记，因为标记必须是一个整数。 方法类似这样：

```
$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end
```

既然有一个整数代表你的提交，那还要给提交元数据一个日期。 因为目录名字表达了日期，所以你将会从中解析出日期。 你的 `print_export` 文件的下一行是

```
date = convert_dir_to_date(dir)
```

`convert_dir_to_date` 定义为

```
def convert_dir_to_date(dir)
  if dir == 'current'
    return Time.now().to_i
  else
    dir = dir.gsub('back_', '')
    (year, month, day) = dir.split('_')
    return Time.local(year, month, day).to_i
  end
end
```

那会返回每一个目录日期的整数。 最后一项每个提交需要的元数据是提交者信息，它将会被硬编码在全局变量中：

```
$author = 'John Doe <john@example.com>'
```

现在准备开始为你的导入器打印出提交数据。 初始信息声明定义了一个提交对象与它所在的分支，紧接着一个你生成的标记、提交者信息与提交信息、然后是一个之前的提交，如果它存在的话。 代码看起来像这样：

```
# print the import information
puts 'commit refs/heads/master'
puts 'mark :' + mark
puts "committer #{$author} #{date} -0700"
export_data('imported from ' + dir)
puts 'from :' + last_mark if last_mark
```

我们将硬编码时区信息（ -0700 ），因为这样很容易。 如果从其他系统导入，必须指定为一个偏移的时区。 提交信息必须指定为特殊的格式：

```
data (size)\n(contents)
```

这个格式包括文本数据、将要读取数据的大小、一个换行符、最终的数据。 因为之后还需要为文件内容指定相同的数据格式，你需要创建一个帮助函数，`export_data`：

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

剩下的工作就是指定每一个快照的文件内容。 这很轻松，因为每一个目录都是一个快照 - 可以在目录中的每一个文件内容后打印 `deleteall` 命令。 Git 将会适当地记录每一个快照：

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

注意：因为大多数系统认为他们的版本是从一个提交变化到另一个提交，fast-import 也可以为每一个提交执行命令来指定哪些文件是添加的、删除的或修改的与新内容是哪些。 可以计算快照间的不同并只提供这些数据，但是这样做会很复杂 - 也可以把所有数据给 Git 然后让它为你指出来。 如果这更适合你的数据，查阅 `fast-import` man 帮助页来了解如何以这种方式提供你的数据。

这种列出新文件内容或用新内容指定修改文件的格式如同下面的内容：

```
M 644 inline path/to/file
data (size)
(file contents)
```

这里，644 是模式（如果你有可执行文件，反而你需要检测并指定 755），inline 表示将会立即把内容放在本行之后。 你的 `inline_data` 方法看起来像这样：

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

可以重用之前定义的 `export_data` 方法，因为它与你定义的提交信息数据的方法一样。

最后一件你需要做的是返回当前的标记以便它可以传给下一个迭代：

```
return mark
```

NOTE

如果在 Windows 上还需要确保增加一个额外步骤。 正如之前提到的，Windows 使用 CRLF 作为换行符而 git fast-import 只接受 LF。 为了修正这个问题使 git fast-import 正常工作，你需要告诉 ruby 使用 LF 代替 CRLF：

```
$stdout.binmode
```

就是这样。 这是全部的脚本：

```
#!/usr/bin/env ruby

$stdout.binmode
$author = "John Doe <john@example.com>"

$marks = []
def convert_dir_to_mark(dir)
    if !$marks.include?(dir)
        $marks << dir
    end
    ($marks.index(dir)+1).to_s
end


def convert_dir_to_date(dir)
    if dir == 'current'
        return Time.now().to_i
    else
        dir = dir.gsub('back_', '')
        (year, month, day) = dir.split('_')
        return Time.local(year, month, day).to_i
    end
end

def export_data(string)
    print "data #{string.size}\n#{string}"
end

def inline_data(file, code='M', mode='644')
    content = File.read(file)
    puts "#{code} #{mode} inline #{file}"
    export_data(content)
end

def print_export(dir, last_mark)
    date = convert_dir_to_date(dir)
    mark = convert_dir_to_mark(dir)

    puts 'commit refs/heads/master'
    puts "mark :#{mark}"
    puts "committer #{$author} #{date} -0700"
    export_data("imported from #{dir}")
    puts "from :#{last_mark}" if last_mark

    puts 'deleteall'
    Dir.glob("**/*").each do |file|
        next if !File.file?(file)
        inline_data(file)
```

```
    end
    mark
end


# Loop through the directories
last_mark = nil
Dir.chdir(ARGV[0]) do
    Dir.glob("*").each do |dir|
        next if File.file?(dir)

        # move into the target directory
        Dir.chdir(dir) do
            last_mark = print_export(dir, last_mark)
        end
    end
end
```

如果运行这个脚本，你会得到类似下面的内容：

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer John Doe <john@example.com> 1388649600 -0700
data 29
imported from back_2014_01_02deleteall
M 644 inline README.md
data 28
# Hello

This is my readme.
commit refs/heads/master
mark :2
committer John Doe <john@example.com> 1388822400 -0700
data 29
imported from back_2014_01_04from :1
deleteall
M 644 inline main.rb
data 34
#!/bin/env ruby

puts "Hey there"
M 644 inline README.md
(...)
```

为了运行导入器，将这些输出用管道重定向到你想要导入的 Git 目录中的 `git fast-import`。 可以创建一个新的目录并在其中运行 `git init` 作为开始，然后运行你的脚本：

```
$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:
---------------------------------------------------------------------
Alloc'd objects:       5000
Total objects:           13 (         6 duplicates                  )
      blobs  :            5 (         4 duplicates          3 deltas of          5 attempts)
      trees  :            4 (         1 duplicates          0 deltas of          4 attempts)
```

```
      commits:            4 (         1 duplicates          0 deltas of          0 attempts)
      tags   :            0 (         0 duplicates          0 deltas of          0 attempts)
Total branches:           1 (         1 loads     )
      marks:           1024 (         5 unique    )
      atoms:              2
Memory total:          2344 KiB
       pools:          2110 KiB
     objects:           234 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit      = 8589934592
pack_report: pack_used_ctr            =         10
pack_report: pack_mmap_calls          =          5
pack_report: pack_open_windows        =          2 /          2
pack_report: pack_mapped              =       1457 /       1457
---------------------------------------------------------------------
```

正如你所看到的，当它成功完成时，它会给你一串关于它完成内容的统计。 这本例中，一共导入了 13 个对象、4 次提交到 1 个分支。 现在，可以运行 `git log` 来看一下你的新历史：

```
$ git log -2
commit 3caa046d4aac682a55867132ccdfbe0d3fdee498
Author: John Doe <john@example.com>
Date:   Tue Jul 29 19:39:04 2014 -0700

    imported from current

commit 4afc2b945d0d3c8cd00556fbe2e8224569dc9def
Author: John Doe <john@example.com>
Date:   Mon Feb 3 01:00:00 2014 -0700

    imported from back_2014_02_03
```

做得很好 - 一个漂亮、干净的 Git 仓库。 要注意的一点是并没有检出任何东西 - 一开始你的工作目录内并没有任何文件。 为了得到他们，你必须将分支重置到 `master` 所在的地方：

```
$ ls
$ git reset --hard master
HEAD is now at 3caa046 imported from current
$ ls
README.md main.rb
```

可以通过 `fast-import` 工具做很多事情 - 处理不同模式、二进制数据、多个分支与合并、标签、进度指示等等。 一些更复杂情形下的例子可以在 Git 源代码目录中的 `contrib/fast-import` 目录中找到。

9.3 总结

总结

你会觉得将 Git 作为其他版本控制系统的客户端，或者在数据无损的情况下将几乎任何一个现有的仓库导入到 Git，都是一件很惬意的事。 在下一章，我们将要讲解 Git 的原始内部数据，如果需要的话你就可以加工每一个字节。

10. Git 内部原理

第10章. Git 内部原理

无论是从之前的章节直接跳到本章，还是读完了其余章节一直到这——你都将在本章见识到 Git 的内部工作原理和实现方式。我们发现学习这部分内容对于理解 Git 的用途和强大至关重要。不过也有人认为这些内容对于初学者而言可能难以理解且过于复杂。因此我们把这部分内容放在最后一章，在学习过程中可以先阅读这部分，也可以晚点阅读这部分，这取决于你自己。

无论如何，既然已经读到了这里，就让我们开始吧。首先要弄明白一点，从根本上来讲 Git 是一个内容寻址（content-addressable）文件系统，并在此之上提供了一个版本控制系统的用户界面。马上你就会学到这意味着什么。

早期的 Git（主要是 1.5 之前的版本）的用户界面要比现在复杂的多，因为它更侧重于作为一个文件系统，而不是一个打磨过的版本控制系统。不时会有一些陈词滥调抱怨早期那个晦涩复杂的 Git 用户界面；不过最近几年来，它已经被改进到不输于任何其他版本控制系统地清晰易用了。

内容寻址文件系统层是一套相当酷的东西，所以在本章我们会先讲解这部分内容。随后我们会学习传输机制和版本库管理任务——你迟早会和它们打交道。

10.1 底层命令和高层命令

底层命令和高层命令

本书旨在讨论如何通过 `checkout`、`branch`、`remote` 等大约 30 个诸如此类动词形式的命令来玩转 Git。 然而，由于 Git 最初是一套面向版本控制系统的工具集，而不是一个完整的、用户友好的版本控制系统，所以它还包含了一部分用于完成底层工作的命令。 这些命令被设计成能以 UNIX 命令行的风格连接在一起，抑或藉由脚本调用，来完成工作。 这部分命令一般被称作“底层（plumbing）”命令，而那些更友好的命令则被称作“高层（porcelain）”命令。

本书前九章专注于探讨高层命令。 然而在本章，我们将主要面对底层命令。 因为，底层命令得以让你窥探 Git 内部的工作机制，也有助于说明 Git 是如何完成工作的，以及它为何如此运作。 多数底层命令并不面向最终用户：它们更适合作为新命令和自定义脚本的组成部分。

当在一个新目录或已有目录执行 `git init` 时，Git 会创建一个 `.git` 目录。 这个目录包含了几乎所有 Git 存储和操作的对象。 如若想备份或复制一个版本库，只需把这个目录拷贝至另一处即可。 本章探讨的所有内容，均位于这个目录内。 该目录的结构如下所示：

```
$ ls -F1
HEAD
config*
description
hooks/
info/
objects/
refs/
```

该目录下可能还会包含其他文件，不过对于一个全新的 `git init` 版本库，这将是你看到的默认结构。 `description` 文件仅供 GitWeb 程序使用，我们无需关心。 `config` 文件包含项目特有的配置选项。 `info` 目录包含一个全局性排除（global exclude）文件，用以放置那些不希望被记录在 .gitignore 文件中的忽略模式（ignored patterns）。 `hooks` 目录包含客户端或服务端的钩子脚本（hook scripts），在 Git 钩子 中这部分话题已被详细探讨过。

剩下的四个条目很重要：`HEAD` 文件、（尚待创建的）`index` 文件，和 `objects` 目录、`refs` 目录。 这些条目是 Git 的核心组成部分。 `objects` 目录存储所有数据内容；`refs` 目录存储指向数据（分支）的提交对象的指针；`HEAD` 文件指示目前被检出的分支；`index` 文件保存暂存区信息。 我们将详细地逐一检视这四部分，以期理解 Git 是如何运转的。

10.2 Git 对象

Git 对象

Git 是一个内容寻址文件系统。 看起来很酷， 但这是什么意思呢？ 这意味着，Git 的核心部分是一个简单的键值对数据库（key-value data store）。 你可以向该数据库插入任意类型的内容，它会返回一个键值，通过该键值可以在任意时刻再次检索（retrieve）该内容。 可以通过底层命令 `hash-object` 来演示上述效果——该命令可将任意数据保存于 `.git` 目录，并返回相应的键值。 首先，我们需要初始化一个新的 Git 版本库，并确认 `objects` 目录为空：

```
$ git init test
Initialized empty Git repository in /tmp/test/.git/
$ cd test
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
```

可以看到 Git 对 `objects` 目录进行了初始化，并创建了 `pack` 和 `info` 子目录，但均为空。 接着，往 Git 数据库存入一些文本：

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

`-w` 选项指示 `hash-object` 命令存储数据对象；若不指定此选项，则该命令仅返回对应的键值。 `--stdin` 选项则指示该命令从标准输入读取内容；若不指定此选项，则须在命令尾部给出待存储文件的路径。 该命令输出一个长度为 40 个字符的校验和。 这是一个 SHA-1 哈希值——一个将待存储的数据外加一个头部信息（header）一起做 SHA-1 校验运算而得的校验和。后文会简要讨论该头部信息。 现在我们可以查看 Git 是如何存储数据的：

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

可以在 `objects` 目录下看到一个文件。 这就是开始时 Git 存储内容的方式——一个文件对应一条内容，以该内容加上特定头部信息一起的 SHA-1 校验和为文件命名。 校验和的前两个字符用于命名子目录，余下的 38 个字符则用作文件名。

可以通过 `cat-file` 命令从 Git 那里取回数据。 这个命令简直就是一把剖析 Git 对象的瑞士军刀。 为 `cat-file` 指定 `-p` 选项可指示该命令自动判断内容的类型，并为我们显示格式友好的内容：

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

至此，你已经掌握了如何向 Git 中存入内容，以及如何将它们取出。 我们同样可以将这些操作应用于文件中的内容。 例如，可以对一个文件进行简单的版本控制。 首先，创建一个新文件并将其内容存入数据库：

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

接着，向文件里写入新内容，并再次将其存入数据库：

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

数据库记录下了该文件的两个不同版本，当然之前我们存入的第一条内容也还在：

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

现在可以把文件内容恢复到第一个版本：

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

或者第二个版本：

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

然而，记住文件的每一个版本所对应的 SHA-1 值并不现实；另一个问题是，在这个（简单的版本控制）系统中，文件名并没有被保存——我们仅保存了文件的内容。 上述类型的对象我们称之为数据对象（blob object）。 利用 `cat-file -t` 命令，可以让 Git 告诉我们其内部存储的任何对象类型，只要给定该对象的 SHA-1 值：

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

```
blob
```

树对象

接下来要探讨的对象类型是树对象（tree object），它能解决文件名保存的问题，也允许我们将多个文件组织到一起。 Git 以一种类似于 UNIX 文件系统的方式存储内容，但作了些许简化。 所有内容均以树对象和数据对象的形式存储，其中树对象对应了 UNIX 中的目录项，数据对象则大致上对应了 inodes 或文件内容。 一个树对象包含了一条或多条树对象记录（tree entry），每条记录含有一个指向数据对象或者子树对象的 SHA-1 指针，以及相应的模式、类型、文件名信息。 例如，某项目当前对应的最新树对象可能是这样的：

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

`master^{tree}` 语法表示 `master` 分支上最新的提交所指向的树对象。 请注意，`lib` 子目录（所对应的那条树对象记录）并不是一个数据对象，而是一个指针，其指向的是另一个树对象：

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

从概念上讲，Git 内部存储的数据有点像这样：

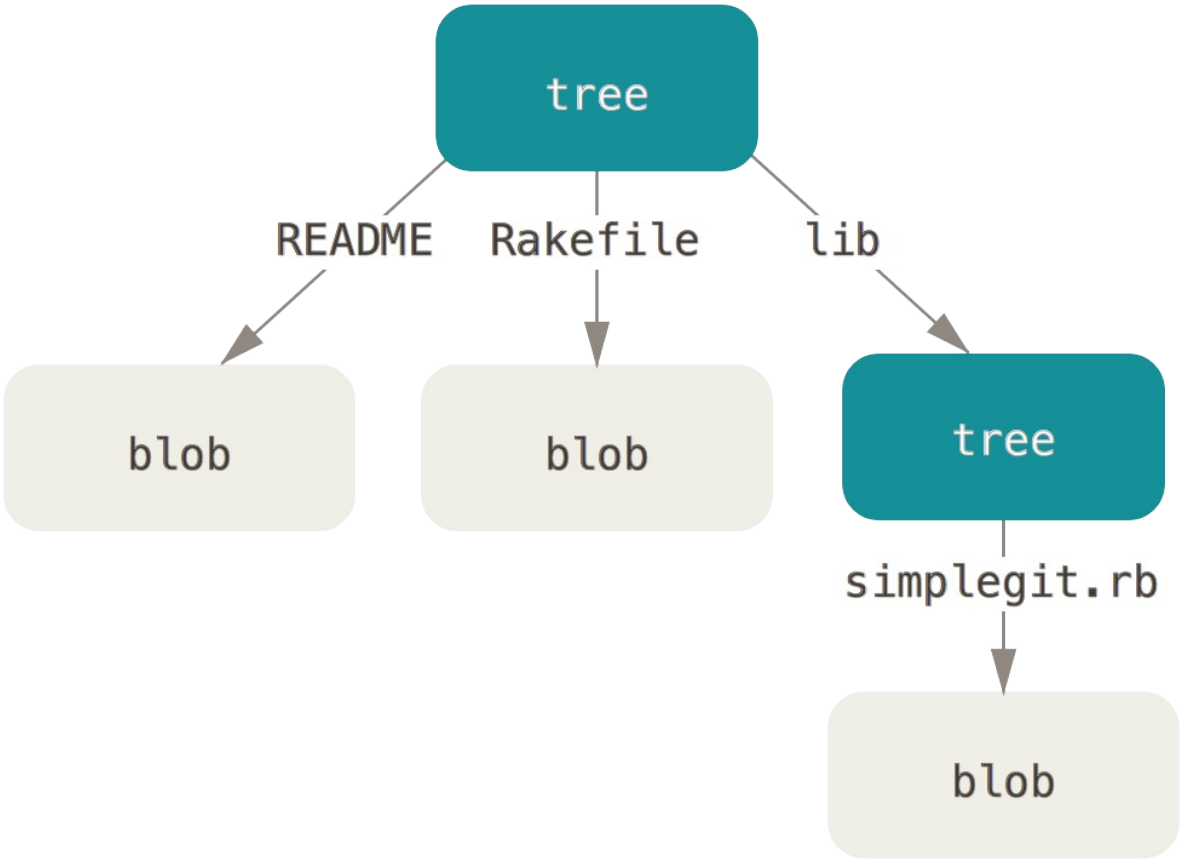


Figure 10-1. 简化版的 Git 数据模型。

你可以轻松创建自己的树对象。 通常，Git 根据某一时刻暂存区（即 index 区域，下同）所表示的状态创建并记录一个对应的树对象，如此重复便可依次记录（某个时间段内）一系列的树对象。 因此，为创建一个树对象，首先需要通过暂存一些文件来创建一个暂存区。 可以通过底层命令 `update-index` 为一个单独文件——我们的 test.txt 文件的首个版本——创建一个暂存区。 利用该命令，可以把 test.txt 文件的首个版本人为地加入一个新的暂存区。 必须为上述命令指定 `--add` 选项，因为此前该文件并不在暂存区中（我们甚至都还没来得及创建一个暂存区呢）；同样必需的还有 `--cacheinfo` 选项，因为将要添加的文件位于 Git 数据库中，而不是位于当前目录下。 同时，需要指定文件模式、SHA-1 与文件名：

```
$ git update-index --add --cacheinfo 100644 \
  83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

本例中，我们指定的文件模式为 `100644`，表明这是一个普通文件。 其他选择包括：`100755`，表示一个可执行文件；`120000`，表示一个符号链接。 这里的文件模式参考了常见的 UNIX 文件模式，但远没那么灵活——上述三种模式即是 Git 文件（即数据对象）的所有合法模式（当然，还有其他一些模式，但用于目录项和子模块）。

现在，可以通过 `write-tree` 命令将暂存区内容写入一个树对象。 此处无需指定 `-w` 选项——如果某个树对象此前并不存在的话，当调用 `write-tree` 命令时，它会根据当前暂存区状态自动创建一个新的树对象：

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

不妨验证一下它确实是一个树对象：

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

接着我们来创建一个新的树对象，它包括 test.txt 文件的第二个版本，以及一个新的文件：

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

暂存区现在包含了 test.txt 文件的新版本，和一个新文件：new.txt。 记录下这个目录树（将当前暂存区的状态记录为一个树对象），然后观察它的结构：

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

我们注意到，新的树对象包含两条文件记录，同时 test.txt 的 SHA-1 值（`1f7a7a`）是先前值的“第二版”。 只是为了好玩：你可以将第一个树对象加入第二个树对象，使其成为新的树对象的一个子目录。 通过调用 `read-tree` 命令，可以把树对象读入暂存区。 本例中，可以通过对 `read-tree` 指定 `--prefix` 选项，将一个已有的树对象作为子树读入暂存区：

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579      bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

如果基于这个新的树对象创建一个工作目录，你会发现工作目录的根目录包含两个文件以及一个名为 `bak` 的子目录，该子目录包含 test.txt 文件的第一个版本。 可以认为 Git 内部存储着的用于表示上述结

构的数据是这样的：

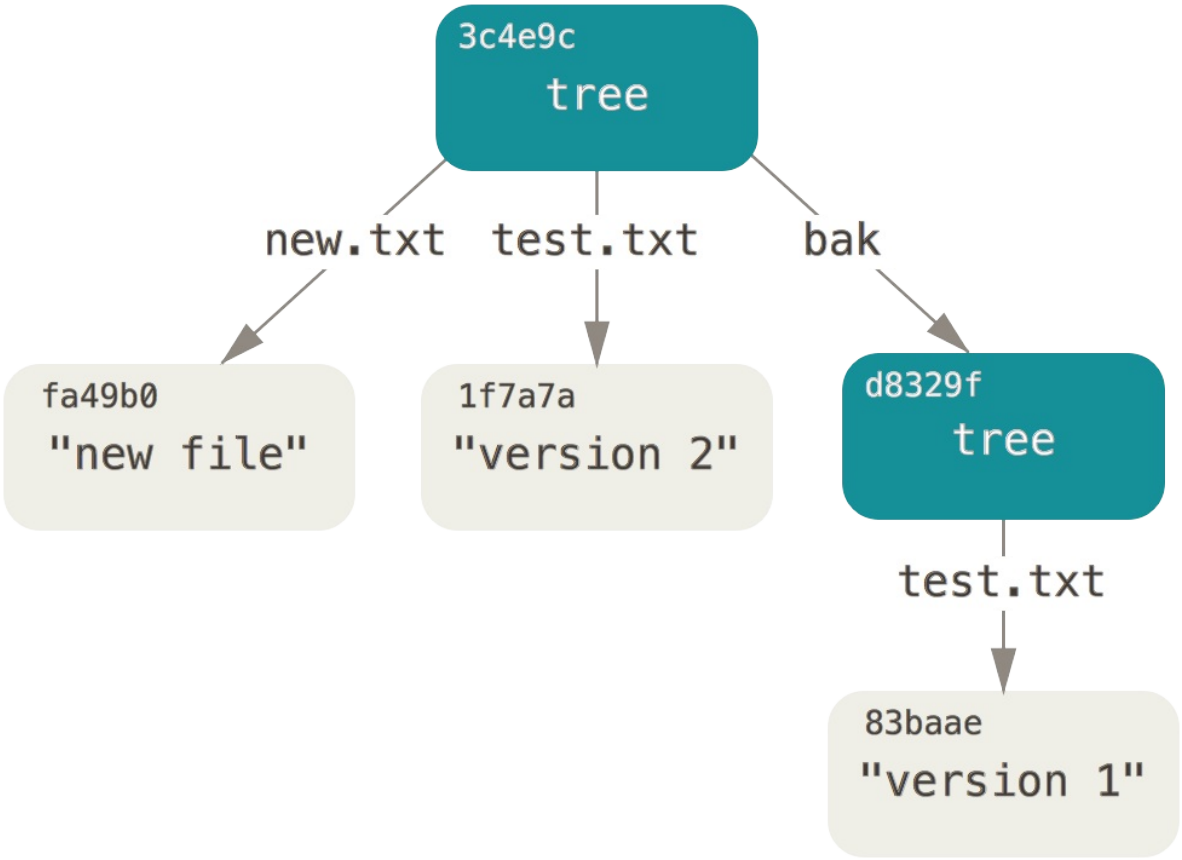


Figure 10-2. 当前 Git 的数据内容结构。

提交对象

现在有三个树对象，分别代表了我们想要跟踪的不同项目快照。然而问题依旧：若想重用这些快照，你必须记住所有三个 SHA-1 哈希值。 并且，你也完全不知道是谁保存了这些快照，在什么时刻保存的，以及为什么保存这些快照。 而以上这些，正是提交对象（commit object）能为你保存的基本信息。

可以通过调用 `commit-tree` 命令创建一个提交对象，为此需要指定一个树对象的 SHA-1 值，以及该提交的父提交对象（如果有的话）。 我们从之前创建的第一个树对象开始：

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

现在可以通过 `cat-file` 命令查看这个新提交对象：

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
```

```
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700

first commit
```

提交对象的格式很简单：它先指定一个顶层树对象，代表当前项目快照；然后是作者/提交者信息（依据你的 `user.name` 和 `user.email` 配置来设定，外加一个时间戳）；留空一行，最后是提交注释。

接着，我们将创建另两个提交对象，它们分别引用各自的上一个提交（作为其父提交对象）：

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit'  | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

这三个提交对象分别指向之前创建的三个树对象快照中的一个。 现在，如果对最后一个提交的 SHA-1 值运行 `git log` 命令，会出乎意料的发现，你已有一个货真价实的、可由 `git log` 查看的 Git 提交历史了：

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

        third commit

 bak/test.txt | 1 +
 1 file changed, 1 insertion(+)

commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:14:29 2009 -0700

        second commit

 new.txt  | 1 +
 test.txt | 2 +-
 2 files changed, 2 insertions(+), 1 deletion(-)

commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:09:34 2009 -0700

    first commit

 test.txt | 1 +
 1 file changed, 1 insertion(+)
```

太神奇了： 就在刚才，你没有借助任何上层命令，仅凭几个底层操作便完成了一个 Git 提交历史的创建。 这就是每次我们运行 `git add` 和 `git commit` 命令时， Git 所做的实质工作——将被改写的文件保存为数据对象，更新暂存区，记录树对象，最后创建一个指明了顶层树对象和父提交的提交对象。 这三种主要的 Git 对象——数据对象、树对象、提交对象——最初均以单独文件的形式保存在 `.git/objects`

目录下。 下面列出了目前示例目录内的所有对象，辅以各自所保存内容的注释：

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

如果跟踪所有的内部指针，将得到一个类似下面的对象关系图：

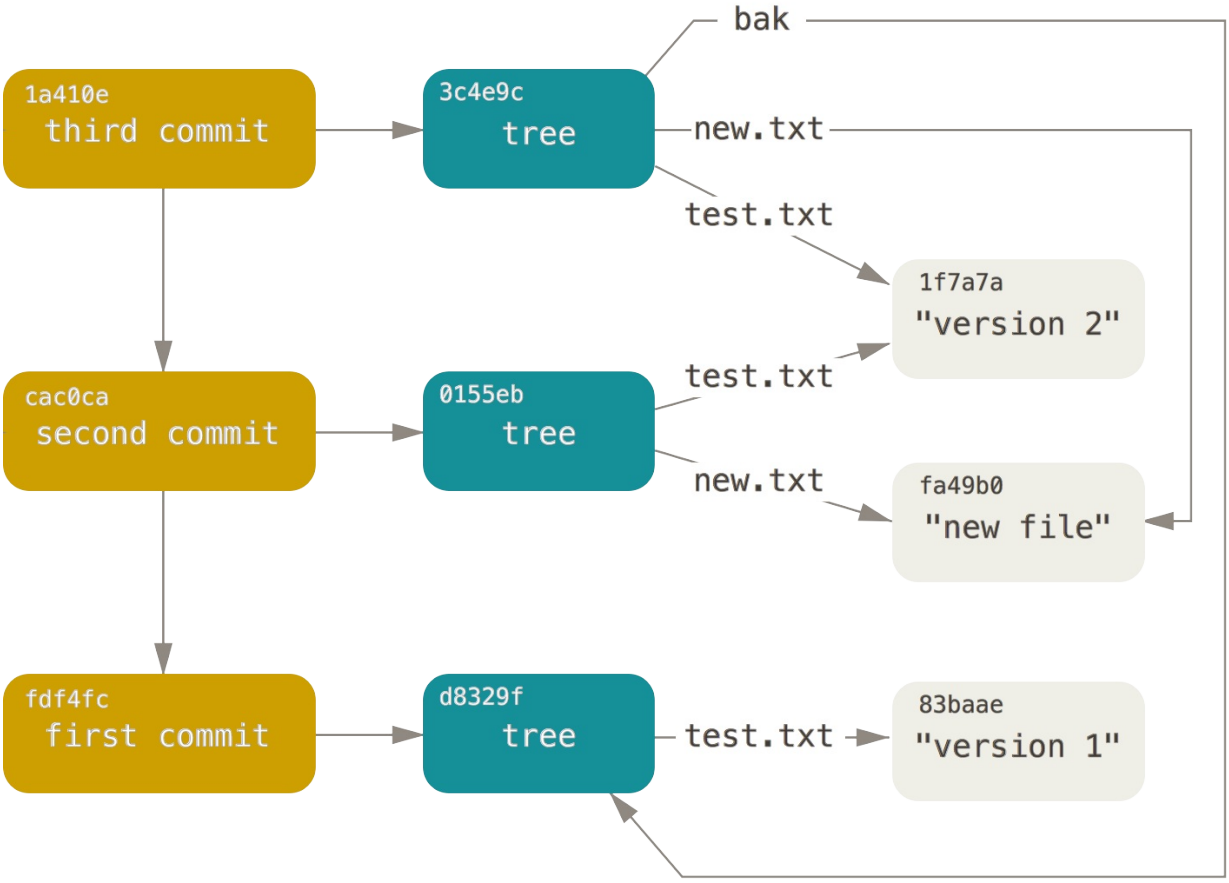


Figure 10-3. 你的 Git 目录下的所有对象。

对象存储

前文曾提及，在存储内容时，会有个头部信息一并被保存。 让我们略花些时间来看看 Git 是如何存储其对象的。 通过在 Ruby 脚本语言中交互式地演示，你将看到一个数据对象——本例中是字符串“what is up, doc?”——是如何被存储的。

可以通过 `irb` 命令启动 Ruby 的交互模式：

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git 以对象类型作为开头来构造一个头部信息，本例中是一个“blob”字符串。 接着 Git 会添加一个空格，随后是数据内容的长度，最后是一个空字节（null byte）：

```
>> header = "blob #{content.length}\0"
=> "blob 16\u0000"
```

Git 会将上述头部信息和原始数据拼接起来，并计算出这条新内容的 SHA-1 校验和。 在 Ruby 中可以这样计算 SHA-1 值——先通过 `require` 命令导入 SHA-1 digest 库，然后对目标字符串调用 `Digest::SHA1.hexdigest()`：

```
>> store = header + content
=> "blob 16\u0000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git 会通过 zlib 压缩这条新内容。在 Ruby 中可以借助 zlib 库做到这一点。 先导入相应的库，然后对目标内容调用 `Zlib::Deflate.deflate()`：

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\x9CK\xCA\xC9OR04c(\xCFH,Q\xC8,V(-\xD0QH\xC9O\xB6\a\x00_\x1C\a\x9D"
```

最后，需要将这条经由 zlib 压缩的内容写入磁盘上的某个对象。 要先确定待写入对象的路径（SHA-1 值的前两个字符作为子目录名称，后 38 个字符则作为子目录内文件的名称）。 如果该子目录不存在，可以通过 Ruby 中的 `FileUtils.mkdir_p()` 函数来创建它。 接着，通过 `File.open()` 打开这个文件。最后，对上一步中得到的文件句柄调用 `write()` 函数，以向目标文件写入之前那条 zlib 压缩过的内容：

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
```

```
=> 32
```

就是这样——你已创建了一个有效的 Git 数据对象。 所有的 Git 对象均以这种方式存储，区别仅在于类型标识——另两种对象类型的头部信息以字符串“commit”或“tree”开头，而不是“blob”。 另外，虽然数据对象的内容几乎可以是任何东西，但提交对象和树对象的内容却有各自固定的格式。

10.3 Git 引用

Git 引用

我们可以借助类似于 `git log 1a410e` 这样的命令来浏览完整的提交历史，但为了能遍历那段历史从而找到所有相关对象，你仍须记住 `1a410e` 是最后一个提交。 我们需要一个文件来保存 SHA-1 值，并给文件起一个简单的名字，然后用这个名字指针来替代原始的 SHA-1 值。

在 Git 里，这样的文件被称为“引用（references，或缩写为 refs）”；你可以在 `.git/refs` 目录下找到这类含有 SHA-1 值的文件。 在目前的项目中，这个目录没有包含任何文件，但它包含了一个简单的目录结构：

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
```

若要创建一个新引用来帮助记忆最新提交所在的位置，从技术上讲我们只需简单地做如下操作：

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

现在，你就可以在 Git 命令中使用这个刚创建的新引用来代替 SHA-1 值了：

```
$ git log --pretty=oneline  master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

我们不提倡直接编辑引用文件。 如果想更新某个引用，Git 提供了一个更加安全的命令 `update-ref` 来完成此事：

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

这基本就是 Git 分支的本质：一个指向某一系列提交之首的指针或引用。 若想在第二个提交上创建一个分支，可以这么做：

```
$ git update-ref refs/heads/test cac0ca
```

这个分支将只包含从第二个提交开始往前追溯的记录：

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

至此，我们的 Git 数据库从概念上看起来像这样：

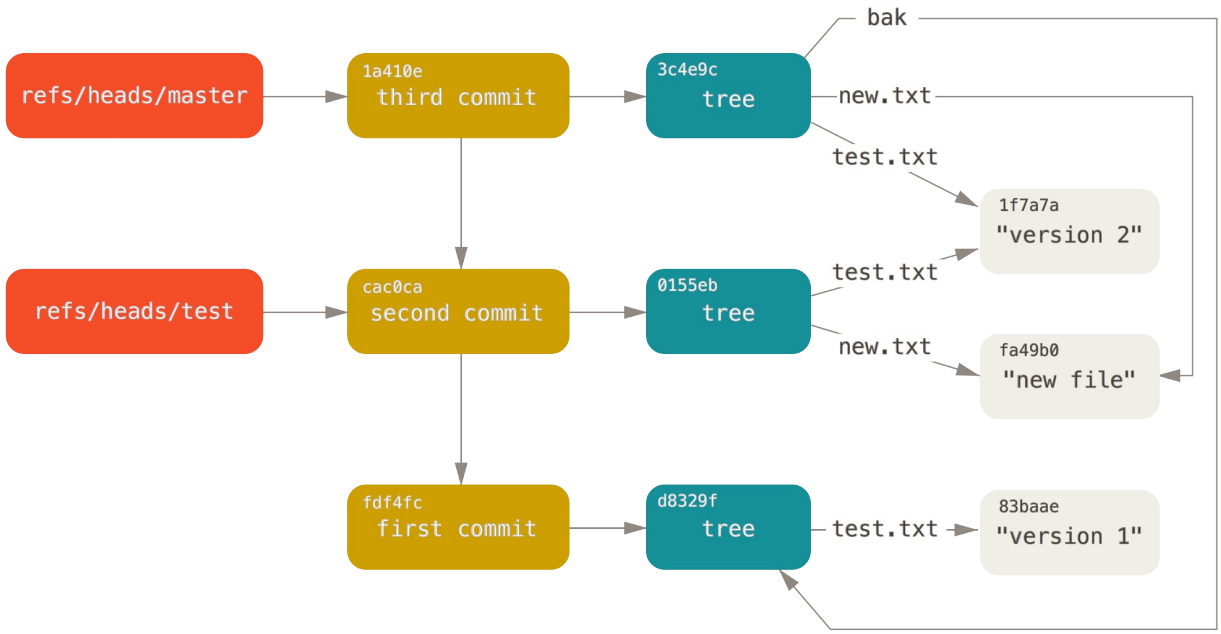


Figure 10-4. 包含分支引用的 Git 目录对象。

当运行类似于 `git branch (branchname)` 这样的命令时，Git 实际上会运行 `update-ref` 命令，取得当前所在分支最新提交对应的 SHA-1 值，并将其加入你想要创建的任何新引用中。

HEAD 引用

现在的问题是，当你执行 `git branch (branchname)` 时，Git 如何知道最新提交的 SHA-1 值呢？答案是 HEAD 文件。

HEAD 文件是一个符号引用（symbolic reference），指向目前所在的分支。 所谓符号引用，意味着它并不像普通引用那样包含一个 SHA-1 值——它是一个指向其他引用的指针。 如果查看 HEAD 文件的内容，一般而言我们看到的类似这样：

```
$ cat .git/HEAD
ref: refs/heads/master
```

如果执行 `git checkout test`，Git 会像这样更新 HEAD 文件：

```
$ cat .git/HEAD
```

```
ref: refs/heads/test
```

当我们执行 `git commit` 时，该命令会创建一个提交对象，并用 HEAD 文件中那个引用所指向的 SHA-1 值设置其父提交字段。

你也可以手动编辑该文件，然而同样存在一个更安全的命令来完成此事：`symbolic-ref`。 可以借助此命令来查看 HEAD 引用对应的值：

```
$ git symbolic-ref HEAD
refs/heads/master
```

同样可以设置 HEAD 引用的值：

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

不能把符号引用设置为一个不符合引用格式的值：

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

标签引用

前文我们刚讨论过 Git 的三种主要对象类型，事实上还有第四种。 标签对象（tag object）非常类似于一个提交对象——它包含一个标签创建者信息、一个日期、一段注释信息，以及一个指针。 主要的区别在于，标签对象通常指向一个提交对象，而不是一个树对象。 它像是一个永不移动的分支引用——永远指向同一个提交对象，只不过给这个提交对象加上一个更友好的名字罢了。

正如 Git 基础 中所讨论的那样，存在两种类型的标签：附注标签和轻量标签。 可以像这样创建一个轻量标签：

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

这就是轻量标签的全部内容——一个固定的引用。 然而，一个附注标签则更复杂一些。 若要创建一个附注标签，Git 会创建一个标签对象，并记录一个引用来指向该标签对象，而不是直接指向提交对象。 可以通过创建一个附注标签来验证这个过程（`-a` 选项指定了要创建的是一个附注标签）：

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

下面是上述过程所建标签对象的 SHA-1 值：

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

现在对该 SHA-1 值运行 `cat-file` 命令：

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cfe9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700

test tag
```

我们注意到，object 条目指向我们打了标签的那个提交对象的 SHA-1 值。 另外要注意的是，标签对象并非必须指向某个提交对象；你可以对任意类型的 Git 对象打标签。 例如，在 Git 源码中，项目维护者将他们的 GPG 公钥添加为一个数据对象，然后对这个对象打了一个标签。 可以克隆一个 Git 版本库，然后通过执行下面的命令来在这个版本库中查看上述公钥：

```
$ git cat-file blob junio-gpg-pub
```

Linux 内核版本库同样有一个不指向提交对象的标签对象——首个被创建的标签对象所指向的是最初被引入版本库的那份内核源码所对应的树对象。

远程引用

我们将看到的第三种引用类型是远程引用（remote reference）。 如果你添加了一个远程版本库并对其执行过推送操作，Git 会记录下最近一次推送操作时每一个分支所对应的值，并保存在 `refs/remotes` 目录下。 例如，你可以添加一个叫做 `origin` 的远程版本库，然后把 `master` 分支推送上去：

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
$ git push origin master
Counting objects: 11, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 716 bytes, done.
Total 7 (delta 2), reused 4 (delta 1)
To git@github.com:schacon/simplegit-progit.git
  a11bef0..ca82a6d  master -> master
```

此时，如果查看 `refs/remotes/origin/master` 文件，可以发现 `origin` 远程版本库的 `master` 分支所对应的 SHA-1 值，就是最近一次与服务器通信时本地 `master` 分支所对应的 SHA-1 值：

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

远程引用和分支（位于 `refs/heads` 目录下的引用）之间最主要的区别在于，远程引用是只读的。 虽然可以 `git checkout` 到某个远程引用，但是 Git 并不会将 HEAD 引用指向该远程引用。因此，你永远不能通过 `commit` 命令来更新远程引用。 Git 将这些远程引用作为记录远程服务器上各分支最后已知位置状态的书签来管理。

10.4 包文件

包文件

让我们重新回到示例 Git 版本库的对象数据库。 目前为止，可以看到有 11 个对象——4 个数据对象、3 个树对象、3 个提交对象和 1 个标签对象：

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Git 使用 zlib 压缩这些文件的内容，而且我们并没有存储太多东西，所以上文中的文件一共只占用了 925 字节。 接下来，我们会指引你添加一些大文件到版本库中，以此展示 Git 的一个很有趣的功能。 为了便于展示，我们要把之前在 Grit 库中用到过的 `repo.rb` 文件添加进来——这是一个大小约为 22K 的源代码文件：

```
$ curl https://raw.githubusercontent.com/mojombo/grit/master/lib/grit/repo.rb > repo.rb
$ git add repo.rb
$ git commit -m 'added repo.rb'
[master 484a592] added repo.rb
 3 files changed, 709 insertions(+), 2 deletions(-)
 delete mode 100644 bak/test.txt
 create mode 100644 repo.rb
 rewrite test.txt (100%)
```

如果你查看生成的树对象，可以看到 repo.rb 文件对应的数据对象的 SHA-1 值：

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

接下来你可以使用 `git cat-file` 命令查看这个对象有多大：

```
$ git cat-file -s 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5
22044
```

现在，稍微修改这个文件，然后看看会发生什么：

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master 2431da6] modified repo.rb a bit
 1 file changed, 1 insertion(+)
```

查看这个提交生成的树对象，你会看到一些有趣的东西：

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob b042a60ef7dff760008df33cee372b945b6e884e      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

repo.rb 对应一个与之前完全不同的数据对象，这意味着，虽然你只是在一个 400 行的文件后面加入一行新内容，Git 也会用一个全新的对象来存储新的文件内容：

```
$ git cat-file -s b042a60ef7dff760008df33cee372b945b6e884e
22054
```

你的磁盘上现在有两个几乎完全相同、大小均为 22K 的对象。 如果 Git 只完整保存其中一个，再保存另一个对象与之前版本的差异内容，岂不更好？

事实上 Git 可以那样做。 Git 最初向磁盘中存储对象时所使用的格式被称为“松散（loose）”对象格式。 但是，Git 会时不时地将多个这些对象打包成一个称为“包文件（packfile）”的二进制文件，以节省空间和提高效率。 当版本库中有太多的松散对象，或者你手动执行 `git gc` 命令，或者你向远程服务器执行推送时，Git 都会这样做。 要看到打包过程，你可以手动执行 `git gc` 命令让 Git 对对象进行打包：

```
$ git gc
Counting objects: 18, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (18/18), done.
Total 18 (delta 3), reused 0 (delta 0)
```

这个时候再查看 objects 目录，你会发现大部分的对象都不见了，与此同时出现了一对新文件：

```
$ find .git/objects -type f
.git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack
```

仍保留着的几个对象是未被任何提交记录引用的数据对象——在此例中是你之前创建的“what is up, doc?”和“test content”这两个示例数据对象。 因为你从没将它们添加至任何提交记录中，所以 Git 认为它们是摇摆（dangling）的，不会将它们打包进新生成的包文件中。

剩下的文件是新创建的包文件和一个索引。 包文件包含了刚才从文件系统中移除的所有对象的内容。 索引文件包含了包文件的偏移信息，我们通过索引文件就可以快速定位任意一个指定对象。 有意思的是运行 `gc` 命令前磁盘上的对象大小约为 22K，而这个新生成的包文件大小仅有 7K。 通过打包对象减少了 ⅔ 的磁盘占用空间。

Git 是如何做到这点的？ Git 打包对象时，会查找命名及大小相近的文件，并只保存文件不同版本之间的差异内容。 你可以查看包文件，观察它是如何节省空间的。 `git verify-pack` 这个底层命令可以让你查看已打包的内容：

```
$ git verify-pack -v .git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx
2431da676938450a4d72e260db3bf7b0f587bbc1 commit 223 155 12
69bcdaff5328278ab1c0812ce0e07fa7d26a96d7 commit 214 152 167
80d02664cb23ed55b226516648c7ad5d0a3deb90 commit 214 145 319
43168a18b7613d1281e5560855a83eb8fde3d687 commit 213 146 464
092917823486a802e94d727c820a9024e14a1fc2 commit 214 146 610
702470739ce72005e2edff522fde85d52a65df9b commit 165 118 756
d368d0ac0678cbe6cce505be58126d3526706e54 tag    130 122 874
fe879577cb8cffcdf25441725141e310dd7d239b tree   136 136 996
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree   36 46 1132
deef2e1b793907545e50a2ea2ddb5ba6c58c4506 tree   136 136 1178
d982c7cb2c2a972ee391a85da481fc1f9127a01d tree   6 17 1314 1 \
  deef2e1b793907545e50a2ea2ddb5ba6c58c4506
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree   8 19 1331 1 \
  deef2e1b793907545e50a2ea2ddb5ba6c58c4506
0155eb4229851634a0f03eb265b69f5a2d56f341 tree   71 76 1350
83baae61804e65cc73a7201a7252750c76066a30 blob   10 19 1426
fa49b077972391ad58037050f2a75f74e3671e92 blob   9 18 1445
b042a60ef7dff760008df33cee372b945b6e884e blob   22054 5799 1463
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob   9 20 7262 1 \
  b042a60ef7dff760008df33cee372b945b6e884e
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob   10 19 7282
non delta: 15 objects
chain length = 1: 3 objects
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack: ok
```

此处，`033b4` 这个数据对象（即 repo.rb 文件的第一个版本，如果你还记得的话）引用了数据对象 `b042a`，即该文件的第二个版本。 命令输出内容的第三列显示的是各个对象在包文件中的大小，可以看到 `b042a` 占用了 22K 空间，而 `033b4` 仅占用 9 字节。 同样有趣的地方在于，第二个版本完整保存了文件内容，而原始的版本反而是以差异方式保存的——这是因为大部分情况下需要快速访问文件的最新版本。

最妙之处是你可以随时重新打包。 Git 时常会自动对仓库进行重新打包以节省空间。当然你也可以随时手动执行 `git gc` 命令来这么做。

10.5 引用规格

引用规格

纵观全书，我们已经使用过一些诸如远程分支到本地引用的简单映射方式，但这种映射可以更复杂。 假设你添加了这样一个远程版本库：

```
$ git remote add origin https://github.com/schacon/simplegit-progit
```

上述命令会在你的 `.git/config` 文件中添加一个小节，并在其中指定远程版本库的名称（ `origin` ）、URL 和一个用于获取操作的引用规格（refspec）：

```
[remote "origin"]
```

引用规格的格式由一个可选的 `+` 号和紧随其后的 `<src>:<dst>` 组成，其中 `<src>` 是一个模式（pattern），代表远程版本库中的引用； `<dst>` 是那些远程引用在本地所对应的位置。 `+` 号告诉 Git 即使在不能快进的情况下也要（强制）更新引用。

默认情况下，引用规格由 `git remote add` 命令自动生成， Git 获取服务器中 `refs/heads/` 下面的所有引用，并将它写入到本地的 `refs/remotes/origin/` 中。 所以，如果服务器上有一个 `master` 分支，我们可以在本地通过下面这种方式来访问该分支上的提交记录：

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

上面的三个命令作用相同，因为 Git 会把它们都扩展成 `refs/remotes/origin/master` 。

如果想让 Git 每次只拉取远程的 `master` 分支，而不是所有分支，可以把（引用规格的）获取那一行修改为：

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

这仅是针对该远程版本库的 `git fetch` 操作的默认引用规格。 如果有某些只希望被执行一次的操作，我们也可以在命令行指定引用规格。 若要将远程的 `master` 分支拉到本地的 `origin/mymaster` 分支，可以运行：

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

你也可以指定多个引用规格。 在命令行中，你可以按照如下的方式拉取多个分支：

```
$ git fetch origin master:refs/remotes/origin/mymaster \
         topic:refs/remotes/origin/topic
From git@github.com:schacon/simplegit
 ! [rejected]        master     -> origin/mymaster  (non fast forward)
 * [new branch]      topic      -> origin/topic
```

在这个例子中，对 `master` 分支的拉取操作被拒绝，因为它不是一个可以快进的引用。 我们可以通过在引用规格之前指定 `+` 号来覆盖该规则。

你也可以在配置文件中指定多个用于获取操作的引用规格。 如果想在每次获取时都包括 `master` 和 `experiment` 分支，添加如下两行：

```
[remote "origin"]
```

我们不能在模式中使用部分通配符，所以像下面这样的引用规格是不合法的：

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

但我们可以使用命名空间（或目录）来达到类似目的。 假设你有一个 QA 团队，他们推送了一系列分支，同时你只想要获取 `master` 和 QA 团队的所有分支而不关心其他任何分支，那么可以使用如下配置：

```
[remote "origin"]
```

如果项目的工作流很复杂，有 QA 团队推送分支、开发人员推送分支、集成团队推送并且在远程分支上展开协作，你就可以像这样（在本地）为这些分支创建各自的命名空间，非常方便。

引用规格推送

像上面这样从远程版本库获取已在命名空间中的引用当然很棒，但 QA 团队最初应该如何将他们的分支放入远程的 `qa/` 命名空间呢？ 我们可以通过引用规格推送来完成这个任务。

如果 QA 团队想把他们的 `master` 分支推送到远程服务器的 `qa/master` 分支上，可以运行：

```
$ git push origin master:refs/heads/qa/master
```

如果他们希望 Git 每次运行 `git push origin` 时都像上面这样推送，可以在他们的配置文件中添加一条 `push` 值：

```
[remote "origin"]
```

正如刚才所指出的，这会让 `git push origin` 默认把本地 `master` 分支推送到远程 `qa/master` 分支。

删除引用

你还可以借助类似下面的命令通过引用规格从远程服务器上删除引用：

```
$ git push origin :topic
```

因为引用规格（的格式）是 `<src>:<dst>`，所以上述命令把 `<src>` 留空，意味着把远程版本库的 `topic` 分支定义为空值，也就是删除它。

10.6 传输协议

传输协议

Git 可以通过两种主要的方式在版本库之间传输数据：“哑（dumb）”协议和“智能（smart）”协议。本节将会带你快速浏览这两种协议的运作方式。

哑协议

如果你正在架设一个基于 HTTP 协议的只读版本库，一般而言这种情况下使用的就是哑协议。 这个协议之所以被称为“哑”协议，是因为在传输过程中，服务端不需要有针对 Git 特有的代码；抓取过程是一系列 HTTP 的 `GET` 请求，这种情况下，客户端可以推断出服务端 Git 仓库的布局。

> NOTE
>
> 现在已经很少使用哑协议了。 使用哑协议的版本库很难保证安全性和私有化，所以大多数 Git 服务器宿主（包括云端和本地）都会拒绝使用它。 一般情况下都建议使用智能协议，我们会在后面进行介绍。

让我们通过 simplegit 版本库来看看 `http-fetch` 的过程：

```
$ git clone http://server/simplegit-progit.git
```

它做的第一件事就是拉取 `info/refs` 文件。 这个文件是通过 `update-server-info` 命令生成的，这也解释了在使用HTTP传输时，必须把它设置为 `post-receive` 钩子的原因：

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949     refs/heads/master
```

现在，你得到了一个远程引用和 SHA-1 值的列表。 接下来，你要确定 HEAD 引用是什么，这样你就知道在完成后应该被检出到工作目录的内容：

```
=> GET HEAD
ref: refs/heads/master
```

这说明在完成抓取后，你需要检出 `master` 分支。 这时，你就可以开始遍历处理了。 因为你是从 `info/refs` 文件中所提到的 `ca82a6` 提交对象开始的，所以你的首要操作是获取它：

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
```

```
(179 bytes of binary data)
```

你取回了一个对象——这是一个在服务端以松散格式保存的对象，是你通过使用静态 HTTP GET 请求获取的。 你可以使用 zlib 解压缩它，去除其头部，查看提交记录的内容：

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

接下来，你还要再获取两个对象，一个是树对象 `cfda3b`，它包含有我们刚刚获取的提交对象所指向的内容，另一个是它的父提交 `085bb3`：

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

这样就取得了你的下一个提交对象。 再抓取树对象：

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

噢——看起来这个树对象在服务端并不以松散格式对象存在，所以你得到了一个 404 响应，代表在 HTTP 服务端没有找到该对象。 这有好几个可能的原因——这个对象可能在替代版本库里面，或者在包文件里面。 Git 会首先检查所有列出的替代版本库：

```
=> GET objects/info/http-alternates
(empty file)
```

如果这返回了一个包含替代版本库 URL 的列表，那么 Git 就会去那些地址检查松散格式对象和文件——这是一种能让派生项目共享对象以节省磁盘的好方法。 然而，在这个例子中，没有列出可用的替代版本库。所以你所需要的对象肯定在某个包文件中。 要检查服务端有哪些可用的包文件，你需要获取 `objects/info/packs` 文件，这里面有一个包文件列表（它也是通过执行 `update-server-info` 所生成的）：

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

服务端只有一个包文件，所以你要的对象显然就在里面。但是你要先检查它的索引文件以确认。 即使服务端有多个包文件，这也是很有用的，因为这样你就可以知道你所需要的对象是在哪一个包文件里面：

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

现在你有这个包文件的索引，你可以查看你要的对象是否在里面——因为索引文件列出了这个包文件所包含的所有对象的 SHA-1 值，和该对象存在于包文件中的偏移量。 你的对象就在这里，接下来就是获取整个包文件：

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k of binary data)
```

现在你也有了你的树对象，你可以继续在提交记录上漫游。 它们全部都在这个你刚下载的包文件里面，所以你不用继续向服务端请求更多下载了。 Git 会将开始时下载的 HEAD 引用所指向的 `master` 分支检出到工作目录。

智能协议

哑协议虽然很简单但效率略低，且它不能从客户端向服务端发送数据。 智能协议是更常用的传送数据的方法，但它需要在服务端运行一个进程，而这也是 Git 的智能之处——它可以读取本地数据，理解客户端有什么和需要什么，并为它生成合适的包文件。 总共有两组进程用于传输数据，它们分别负责上传和下载数据。

上传数据

为了上传数据至远端，Git 使用 `send-pack` 和 `receive-pack` 进程。 运行在客户端上的 `send-pack` 进程连接到远端运行的 `receive-pack` 进程。

SSH

举例来说，在项目中使用命令 `git push origin master` 时，`origin` 是由基于 SSH 协议的 URL 所定义的。 Git 会运行 `send-pack` 进程，它会通过 SSH 连接你的服务器。 它会尝试通过 SSH 在服务端执行命令，就像这样：

```
$ ssh -x git@server "git-receive-pack 'simplegit-progit.git'"
00a5ca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status \
        delete-refs side-band-64k quiet ofs-delta \
        agent=git/2:2.1.1+github-607-gfba4028 delete-refs
0000
```

`git-receive-pack` 命令会立即为它所拥有的每一个引用发送一行响应——在这个例子中，就只有 `master` 分支和它的 SHA-1 值。 第一行响应中也包含了一个服务端能力的列表（这里是 `report-status`、`delete-refs` 和一些其它的，包括客户端的识别码）。

每一行以一个四位的十六进制值开始，用于指明本行的长度。 你看到第一行以 005b 开始，这在十六进制中表示 91，意味着第一行有 91 字节。 下一行以 003e 起始，也就是 62，所以下面需要读取 62 字节。

再下一行是 0000，表示服务端已完成了发送引用列表过程。

现在它知道了服务端的状态，你的 `send-pack` 进程会判断哪些提交记录是它所拥有但服务端没有的。 `send-pack` 会告知 `receive-pack` 这次推送将会更新的各个引用。 举个例子，如果你正在更新 `master` 分支，并且增加 `experiment` 分支，这个 `send-pack` 的响应将会是像这样：

```
0076ca82a6dff817ec66f44342007202690a93763949 15027957951b64cf874c3557a0f3547bd83b3ff6 \
        refs/heads/master report-status
006c0000000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf63e8d \
        refs/heads/experiment
0000
```

Git 会为每一个将要更新的引用发送一行数据，包括该行长度，旧 SHA-1 值，新 SHA-1 值和将要更新的引用。 第一行也包括了客户端的能力。 这里的全为 0 的 SHA-1 值表示之前没有过这个引用——因为你正要添加新的 experiment 引用。 删除引用时，将会看到相反的情况：右边的 SHA-1 值全为 0。

接下来，客户端会发送一个包文件，它包含了所有服务端还没有的对象。 最后，服务端会以成功（或失败）响应：

```
000eunpack ok
```

HTTP(S)
HTTPS 与 HTTP 相比较，除了在“握手”过程略有不同外，其他基本相似。 连接是从下面这个请求开始的：

```
=> GET http://server/simplegit-progit.git/info/refs?service=git-receive-pack
001f# service=git-receive-pack
00ab6c5f0e45abd7832bf23074a333f739977c9e8188 refs/heads/master report-status \
        delete-refs side-band-64k quiet ofs-delta \
        agent=git/2:2.1.1~vmg-bitmaps-bugaloo-608-g116744e
0000
```

这完成了客户端和服务端的第一次数据交换。 接下来客户端发起另一个请求，这次是一个 `POST` 请求，这个请求中包含了 `git-upload-pack` 提供的数据。

```
=> POST http://server/simplegit-progit.git/git-receive-pack
```

这个 `POST` 请求的内容是 `send-pack` 的输出和相应的包文件。 服务端在收到请求后相应地作出成功或失败的 HTTP 响应。

下载数据
当你在下载数据时， `fetch-pack` 和 `upload-pack` 进程就起作用了。 客户端启动 `fetch-pack`

进程，连接至远端的 `upload-pack` 进程，以协商后续传输的数据。

SSH
如果你通过 SSH 使用抓取功能， `fetch-pack` 会像这样运行：

```
$ ssh -x git@server "git-upload-pack 'simplegit-progit.git'"
```

在 `fetch-pack` 连接后， `upload-pack` 会返回类似下面的内容：

```
00dfca82a6dff817ec66f44342007202690a93763949 HEAD multi_ack thin-pack \
        side-band side-band-64k ofs-delta shallow no-progress include-tag \
        multi_ack_detailed symref=HEAD:refs/heads/master \
        agent=git/2:2.1.1+github-607-gfba4028
003fe2409a098dc3e53539a9028a94b6224db9d6a6b6 refs/heads/master
0000
```

这与 `receive-pack` 的响应很相似，但是这里所包含的能力是不同的。 而且它还包含 HEAD 引用所指向内容（ `symref=HEAD:refs/heads/master` ），这样如果客户端执行的是克隆，它就会知道要检出什么。

这时候， `fetch-pack` 进程查看它自己所拥有的对象，并响应 “want” 和它需要的对象的 SHA-1 值。 它还会发送“have”和所有它已拥有的对象的 SHA-1 值。 在列表的最后，它还会发送“done”以通知 `upload-pack` 进程可以开始发送它所需对象的包文件：

```
003cwant ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0009done
0000
```

HTTP(S)
抓取操作的握手需要两个 HTTP 请求。 第一个是向和哑协议中相同的端点发送 `GET` 请求：

```
=> GET $GIT_URL/info/refs?service=git-upload-pack
001e# service=git-upload-pack
00e7ca82a6dff817ec66f44342007202690a93763949 HEAD multi_ack thin-pack \
        side-band side-band-64k ofs-delta shallow no-progress include-tag \
        multi_ack_detailed no-done symref=HEAD:refs/heads/master \
        agent=git/2:2.1.1+github-607-gfba4028
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

这和通过 SSH 使用 `git-upload-pack` 是非常相似的，但是第二个数据交换则是一个单独的请求：

```
=> POST $GIT_URL/git-upload-pack HTTP/1.0
0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7
```

```
0032have 441b40d833fdfa93eb2908e52742248faf0ee993
0000
```

这个输出格式还是和前面一样的。 这个请求的响应包含了所需要的包文件，并指明成功或失败。

协议总结

这一章节是传输协议的一个概貌。 传输协议还有很多其它的特性，像是 `multi_ack` 或 `side-band` ，但是这些内容已经超出了本书的范围。 我们希望能给你展示客户端和服务端之间的基本交互过程；如果你需要更多的相关知识，你可以参阅 Git 的源代码。

10.7 维护与数据恢复

维护与数据恢复

有的时候，你需要对仓库进行清理 - 使它的结构变得更紧凑，或是对导入的仓库进行清理，或是恢复丢失的内容。 这个小节将会介绍这些情况中的一部分。

维护

Git 会不定时地自动运行一个叫做 “auto gc” 的命令。 大多数时候，这个命令并不会产生效果。 然而，如果有太多松散对象（不在包文件中的对象）或者太多包文件，Git 会运行一个完整的 `git gc` 命令。 “gc” 代表垃圾回收，这个命令会做以下事情：收集所有松散对象并将它们放置到包文件中，将多个包文件合并为一个大的包文件，移除与任何提交都不相关的陈旧对象。

可以像下面一样手动执行自动垃圾回收：

```
$ git gc --auto
```

就像上面提到的，这个命令通常并不会产生效果。 大约需要 7000 个以上的松散对象或超过 50 个的包文件才能让 Git 启动一次真正的 gc 命令。 你可以通过修改 `gc.auto` 与 `gc.autopacklimit` 的设置来改动这些数值。

`gc` 将会做的另一件事是打包你的引用到一个单独的文件。 假设你的仓库包含以下分支与标签：

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

如果你执行了 `git gc` 命令，`refs` 目录中将不会再有这些文件。 为了保证效率 Git 会将它们移动到名为 `.git/packed-refs` 的文件中，就像这样：

```
$ cat .git/packed-refs
# pack-refs with: peeled fully-peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

如果你更新了引用，Git 并不会修改这个文件，而是向 `refs/heads` 创建一个新的文件。 为了获得指定引用的正确 SHA-1 值，Git 会首先在 `refs` 目录中查找指定的引用，然后再到 `packed-refs` 文件中

查找。 所以，如果你在 `refs` 目录中找不到一个引用，那么它或许在 `packed-refs` 文件中。

注意这个文件的最后一行，它会以 `^` 开头。 这个符号表示它上一行的标签是附注标签，那一行是附注标签指向的那个提交。

数据恢复

在你使用 Git 的时候，你可能会意外丢失一次提交。 通常这是因为你强制删除了正在工作的分支，但是最后却发现你还需要这个分支；亦或者硬重置了一个分支，放弃了你想要的提交。 如果这些事情已经发生，该如何找回你的提交呢？

下面的例子将硬重置你的测试仓库中的 master 分支到一个旧的提交，以此来恢复丢失的提交。 首先，让我们看看你的仓库现在在什么地方：

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

现在，我们将 `master` 分支硬重置到第三次提交：

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

现在顶部的两个提交已经丢失了 - 没有分支指向这些提交。 你需要找出最后一次提交的 SHA-1 然后增加一个指向它的分支。 窍门就是找到最后一次的提交的 SHA-1 - 但是估计你记不起来了，对吗？

最方便，也是最常用的方法，是使用一个名叫 `git reflog` 的工具。 当你正在工作时，Git 会默默地记录每一次你改变 HEAD 时它的值。 每一次你提交或改变分支，引用日志都会被更新。 引用日志（reflog）也可以通过 `git update-ref` 命令更新，我们在 Git 引用 有提到使用这个命令而不是是直接将 SHA-1 的值写入引用文件中的原因。 你可以在任何时候通过执行 `git reflog` 命令来了解你曾经做过什么：

```
$ git reflog
1a410ef HEAD@{0}: reset: moving to 1a410ef
ab1afef HEAD@{1}: commit: modified repo.rb a bit
484a592 HEAD@{2}: commit: added repo.rb
```

这里可以看到我们已经检出的两次提交，然而并没有足够多的信息。 为了使显示的信息更加有用，我们可以执行 `git log -g` ，这个命令会以标准日志的格式输出引用日志。

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:22:37 2009 -0700

               third commit

commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

       modified repo.rb a bit
```

看起来下面的那个就是你丢失的提交，你可以通过创建一个新的分支指向这个提交来恢复它。 例如，你可以创建一个名为 `recover-branch` 的分支指向这个提交（ab1afef）：

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

不错，现在有一个名为 `recover-branch` 的分支是你的 `master` 分支曾经指向的地方，再一次使得前两次提交可到达了。 接下来，假设你丢失的提交因为某些原因不在引用日志中 - 我们可以通过移除 `recover-branch` 分支并删除引用日志来模拟这种情况。 现在前两次提交又不被任何分支指向了：

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

由于引用日志数据存放在 `.git/logs/` 目录中，现在你已经没有引用日志了。 这时该如何恢复那次提交？ 一种方式是使用 `git fsck` 实用工具，将会检查数据库的完整性。 如果使用一个 `--full` 选项运行它，它会向你显示出所有没有被其他对象指向的对象：

```
$ git fsck --full
Checking object directories: 100% (256/256), done.
Checking objects: 100% (18/18), done.
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

在这个例子中，你可以在 “dangling commit” 后看到你丢失的提交。 现在你可以用和之前相同的方法

恢复这个提交，也就是添加一个指向这个提交的分支。

移除对象

Git 有很多很棒的功能，但是其中一个特性会导致问题，`git clone` 会下载整个项目的历史，包括每一个文件的每一个版本。 如果所有的东西都是源代码那么这很好，因为 Git 被高度优化来有效地存储这种数据。 然而，如果某个人在之前向项目添加了一个大小特别大的文件，即使你将这个文件从项目中移除了，每次克隆还是都要强制的下载这个大文件。 之所以会产生这个问题，是因为这个文件在历史中是存在的，它会永远在那里。

当你迁移 Subversion 或 Perforce 仓库到 Git 的时候，这会是一个严重的问题。 因为这些版本控制系统并不下载所有的历史文件，所以这种文件所带来的问题比较少。 如果你从其他的版本控制系统迁移到 Git 时发现仓库比预期的大得多，那么你就需要找到并移除这些大文件。

警告：这个操作对提交历史的修改是破坏性的。 它会从你必须修改或移除一个大文件引用最早的树对象开始重写每一次提交。 如果你在导入仓库后，在任何人开始基于这些提交工作前执行这个操作，那么将不会有任何问题 - 否则，你必须通知所有的贡献者他们需要将他们的成果变基到你的新提交上。

为了演示，我们将添加一个大文件到测试仓库中，并在下一次提交中删除它，现在我们需要找到它，并将它从仓库中永久删除。 首先，添加一个大文件到仓库中：

```
$ curl https://www.kernel.org/pub/software/scm/git/git-2.1.0.tar.gz > git.tgz
$ git add git.tgz
$ git commit -m 'add git tarball'
[master 7b30847] add git tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tgz
```

哎呀 - 其实这个项目并不需要这个巨大的压缩文件。 现在我们将它移除：

```
$ git rm git.tgz
rm 'git.tgz'
$ git commit -m 'oops - removed large tarball'
[master dadf725] oops - removed large tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tgz
```

现在，我们执行 `gc` 来查看数据库占用了多少空间：

```
$ git gc
Counting objects: 17, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

你也可以执行 `count-objects` 命令来快速的查看占用空间大小：

```
$ git count-objects -v
count: 7
size: 32
in-pack: 17
packs: 1
size-pack: 4868
prune-packable: 0
garbage: 0
size-garbage: 0
```

`size-pack` 的数值指的是你的包文件以 KB 为单位计算的大小，所以你大约占用了 5MB 的空间。 在最后一次提交前，使用了不到 2KB - 显然，从之前的提交中移除文件并不能从历史中移除它。 每一次有人克隆这个仓库时，他们将必须克隆所有的 5MB 来获得这个微型项目，只因为你意外地添加了一个大文件。 现在来让我们彻底的移除这个文件。

首先你必须找到它。 在本例中，你已经知道是哪个文件了。 但是假设你不知道；该如何找出哪个文件或哪些文件占用了如此多的空间？ 如果你执行 `git gc` 命令，所有的对象将被放入一个包文件中，你可以通过运行 `git verify-pack` 命令，然后对输出内容的第三列（即文件大小）进行排序，从而找出这个大文件。 你也可以将这个命令的执行结果通过管道传送给 `tail` 命令，因为你只需要找到列在最后的几个大对象。

```
$ git verify-pack -v .git/objects/pack/pack-29…69.idx \
  | sort -k 3 -n \
  | tail -3
dadf7258d699da2c8d89b09ef6670edb7d5f91b4 commit 229 159 12
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob   22044 5792 4977696
82c99a3e86bb1267b236a4b6eff7868d97489af1 blob   4975916 4976258 1438
```

你可以看到这个大对象出现在返回结果的最底部：占用 5MB 空间。 为了找出具体是哪个文件，可以使用 `rev-list` 命令，我们在 指定特殊的提交信息格式 中曾提到过。 如果你传递 `--objects` 参数给 `rev-list` 命令，它就会列出所有提交的 SHA-1、数据对象的 SHA-1 和与它们相关联的文件路径。 可以使用以下命令来找出你的数据对象的名字：

```
$ git rev-list --objects --all | grep 82c99a3
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

现在，你只需要从过去所有的树中移除这个文件。 使用以下命令可以轻松地查看哪些提交对这个文件产生改动：

```
$ git log --oneline --branches -- git.tgz
dadf725 oops - removed large tarball
7b30847 add git tarball
```

现在，你必须重写 `7b30847` 提交之后的所有提交来从 Git 历史中完全移除这个文件。 为了执行这个操作，我们要使用 `filter-branch` 命令，这个命令在 重写历史 中也使用过：

```
$ git filter-branch --index-filter \
  'git rm --ignore-unmatch --cached git.tgz' -- 7b30847^..
Rewrite 7b30847d080183a1ab7d18fb202473b3096e9f34 (1/2)rm 'git.tgz'
Rewrite dadf7258d699da2c8d89b09ef6670edb7d5f91b4 (2/2)
Ref 'refs/heads/master' was rewritten
```

`--index-filter` 选项类似于在 重写历史 中提到的的 `--tree-filter` 选项，不过这个选项并不会让命令将修改在硬盘上检出的文件，而只是修改在暂存区或索引中的文件。

你必须使用 `git rm --cached` 命令来移除文件，而不是通过类似 `rm file` 的命令 - 因为你需要从索引中移除它，而不是磁盘中。 还有一个原因是速度 - Git 在运行过滤器时，并不会检出每个修订版本到磁盘中，所以这个过程会非常快。 如果愿意的话，你也可以通过 `--tree-filter` 选项来完成同样的任务。 `git rm` 命令的 `--ignore-unmatch` 选项告诉命令：如果尝试删除的模式不存在时，不提示错误。 最后，使用 `filter-branch` 选项来重写自 `7b30847` 提交以来的历史，也就是这个问题产生的地方。 否则，这个命令会从最旧的提交开始，这将会花费许多不必要的时间。

你的历史中将不再包含对那个文件的引用。 不过，你的引用日志和你在 `.git/refs/original` 通过 `filter-branch` 选项添加的新引用中还存有对这个文件的引用，所以你必须移除它们然后重新打包数据库。 在重新打包前需要移除任何包含指向那些旧提交的指针的文件：

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 15, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (15/15), done.
Total 15 (delta 1), reused 12 (delta 0)
```

让我们看看你省了多少空间。

```
$ git count-objects -v
count: 11
size: 4904
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

打包的仓库大小下降到了 8K，比 5MB 好很多。 可以从 size 的值看出，这个大文件还在你的松散对象中，并没有消失；但是它不会在推送或接下来的克隆中出现，这才是最重要的。 如果真的想要删除它，可

以通过有 `--expire` 选项的 `git prune` 命令来完全地移除那个对象：

```
$ git prune --expire now
$ git count-objects -v
count: 0
size: 0
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

10.8 环境变量

环境变量

Git 总是在一个 `bash` shell 中运行，并借助一些 shell 环境变量来决定它的运行方式。 有时候，知道它们是什么以及它们如何让 Git 按照你想要的方式去运行会很有用。 这里不会列出所有的 Git 环境变量，但我们会涉及最有的那部分。

全局行为

像通常的程序一样，Git 的常规行为依赖于环境变量。

`GIT_EXEC_PATH` 决定 Git 到哪找它的子程序（像 `git-commit`，`git-diff` 等等）。 你可以用 `git --exec-path` 来查看当前设置.

通常不会考虑修改 **`HOME`** 这个变量（太多其它东西都依赖它），这是 Git 查找全局配置文件的地方。 如果你想要一个包括全局配置的真正的便携版 Git， 你可以在便携版 Git 的 shell 配置中覆盖 `HOME` 设置。

`PREFIX` 也类似，除了用于系统级别的配置。 Git 在 `$PREFIX/etc/gitconfig` 查找此文件.

如果设置了 **`GIT_CONFIG_NOSYSTEM`**，就禁用系统级别的配置文件。 这在系统配置影响了你的命令，而你又无权限修改的时候很有用。

`GIT_PAGER` 控制在命令行上显示多页输出的程序。 如果这个没有设置，就会用 `PAGER` .

`GIT_EDITOR` 当用户需要编辑一些文本（比如提交信息）时， Git 会启动这个编辑器。 如果没设置，就会用 `EDITOR` 。

版本库位置

Git 用了几个变量来确定它如何与当前版本库交互。

`GIT_DIR` 是 `.git` 目录的位置. 如果这个没有设置， Git 会按照目录树逐层向上查找 `.git` 目录，直到到达 `~` 或 `/` 。

`GIT_CEILING_DIRECTORIES` 控制查找 `.git` 目录的行为。 如果你访问加载很慢的目录（如那些磁带机上的或通过网络连接访问的），你可能会想让 Git 早点停止尝试，尤其是 shell 构建时调用了 Git 。

`GIT_WORK_TREE` 是非空版本库的工作目录的根路径 如果没指定，就使用 `$GIT_DIR` 的父目录。

`GIT_INDEX_FILE` 是索引文件的路径（只有非空版本库有）

`GIT_OBJECT_DIRECTORY` 用来指定 `.git/objects` 目录的位置。

`GIT_ALTERNATE_OBJECT_DIRECTORIES` 一个冒号分割的列表 (格式类似 `/dir/one:/dir/two:…`) 用来告诉 Git 到哪里去找不在 `GIT_OBJECT_DIRECTORY` 目录中的对象. 如果你有很多项目有相同内容的大文件，这个可以用来避免存储过多备份。

路径规则

所谓 “pathspec” 是指你在 Git 中如何指定路径， 包括通配符的使用。 它们会在 `.gitignore` 文件中用到，命令行里也会用到（ `git add *.c` ）。

`GIT_GLOB_PATHSPECS` and `GIT_NOGLOB_PATHSPECS` 控制通配符在路径规则中的默认行为。 如果 `GIT_GLOB_PATHSPECS` 设置为 1, 通配符表现为通配符（这是默认设置）; 如果 `GIT_NOGLOB_PATHSPECS` 设置为 1,通配符仅匹配字面。意思是 `*.c` 只会匹配 文件名是 “.c” 的文件， 而不是以 `.c` 结尾的文件。 你可以在各个路径规格中用 `:(glob)` 或 `:(literal)` 开头来覆盖这个配置，如 `:(glob).c` 。

`GIT_LITERAL_PATHSPECS` 禁用上面的两种行为；通配符将不能用，前缀覆盖也不能用。

`GIT_ICASE_PATHSPECS` 让所有的路径规格忽略大小写。

提交

Git 提交对象的创建通常最后是由 `git-commit-tree` 来完成， `git-commit-tree` 用这些环境变量作主要的信息源。 仅当这些值不存在才回退到预置的值。

`GIT_AUTHOR_NAME` 是 “author” 字段的可读的名字。

`GIT_AUTHOR_EMAIL` 是 “author” 字段的邮件。

`GIT_AUTHOR_DATE` 是 “author” 字段的时间戳。

`GIT_COMMITTER_NAME` 是 “committer” 字段的可读的名字。

`GIT_COMMITTER_EMAIL` 是 “committer” 字段的邮件。

`GIT_COMMITTER_DATE` 是 “committer” 字段的时间戳。

如果 `user.email` 没有配置， 就会用到 `EMAIL` 指定的邮件地址。 如果 这个 也没有设置， Git 继续回退使用系统用户和主机名。

网络

Git 使用 `curl` 库通过 HTTP来完成网络操作， 所以 `GIT_CURL_VERBOSE` 告诉 Git 显示所有由那个库产生的消息。 这跟在命令行执行 `curl -v` 差不多。

`GIT_SSL_NO_VERIFY` 告诉 Git 不用验证 SSL 证书。 这在有些时候是需要的， 例如你用一个自己签名的证书通过 HTTPS 来提供 Git 服务， 或者你正在搭建 Git 服务器，还没有安装完全的证书。

如果 Git 操作在网速低于 `GIT_HTTP_LOW_SPEED_LIMIT` 字节 / 秒，并且持续 `GIT_HTTP_LOW_SPEED_TIME` 秒以上的时间，Git 会终止那个操作。 这些值会覆盖 `http.lowSpeedLimit` 和 `http.lowSpeedTime` 配置的值。

`GIT_HTTP_USER_AGENT` 设置 Git 在通过 HTTP 通讯时用到的 user-agent。 默认值类似于 `git/2.0.0` 。

比较和合并

`GIT_DIFF_OPTS` 这个有点起错名字了 有效值仅支持 `-u<n>` 或 `--unified=<n>` ，用来控制在 `git diff` 命令中显示的内容行数。

`GIT_EXTERNAL_DIFF` 用来覆盖 `diff.external` 配置的值。 如果设置了这个值， 当执行Git `git diff` 时，Git 会调用该程序。

`GIT_DIFF_PATH_COUNTER` 和 `GIT_DIFF_PATH_TOTAL` 对于 `GIT_EXTERNAL_DIFF` 或 `diff.external` 指定的程序有用。 前者表示在一系列文件中哪个是被比较的（从 1 开始），后者表示每批文件的总数。

`GIT_MERGE_VERBOSITY` 控制递归合并策略的输出。 允许的值有下面这些：

- 0 什么都不输出，除了可能会有一个错误信息。
- 1 只显示冲突。
- 2 还显示文件改变。
- 3 显示因为没有改变被跳过的文件。
- 4 显示处理的所有路径。
- 5 显示详细的调试信息。

默认值是 2.

调试

想 真正地 知道 Git 正在做什么? Git 内置了相当完整的跟踪信息，你需要做的就是把它们打开。 这些变量的可以用的值如下：

- “true”， “1”，或 “2” – 跟踪类别写到标准错误输出.
- 以 `/` 开头的绝对路径 – 跟踪输出会被写到那个文件。

`GIT_TRACE` 控制常规跟踪，它并不适用于特殊情况。 它跟踪的范围包括别名的展开和其他子程序的委托。

```
$ GIT_TRACE=true git lga
20:12:49.877982 git.c:554               trace: exec: 'git-lga'
20:12:49.878369 run-command.c:341       trace: run_command: 'git-lga'
20:12:49.879529 git.c:282               trace: alias expansion: lga => 'log' '--graph' '--pretty=onelin
e' '--abbrev-commit' '--decorate' '--all'
20:12:49.879885 git.c:349               trace: built-in: git 'log' '--graph' '--pretty=oneline' '--abbr
ev-commit' '--decorate' '--all'
20:12:49.899217 run-command.c:341       trace: run_command: 'less'
20:12:49.899675 run-command.c:192       trace: exec: 'less'
```

GIT_TRACE_PACK_ACCESS 控制访问打包文件的跟踪信息 第一个字段是被访问的打包文件，第二个是文件的偏移量：

```
$ GIT_TRACE_PACK_ACCESS=true git status
20:10:12.081397 sha1_file.c:2088        .git/objects/pack/pack-c3fa...291e.pack 12
20:10:12.081886 sha1_file.c:2088        .git/objects/pack/pack-c3fa...291e.pack 34662
20:10:12.082115 sha1_file.c:2088        .git/objects/pack/pack-c3fa...291e.pack 35175
# […]
20:10:12.087398 sha1_file.c:2088        .git/objects/pack/pack-e80e...e3d2.pack 56914983
20:10:12.087419 sha1_file.c:2088        .git/objects/pack/pack-e80e...e3d2.pack 14303666
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

GIT_TRACE_PACKET 打开网络操作包级别的跟踪信息

```
$ GIT_TRACE_PACKET=true git ls-remote origin
20:15:14.867043 pkt-line.c:46           packet:          git< # service=git-upload-pack
20:15:14.867071 pkt-line.c:46           packet:          git< 0000
20:15:14.867079 pkt-line.c:46           packet:          git< 97b8860c071898d9e162678ea1035a8ced2f8b1f
HEAD\0multi_ack thin-pack side-band side-band-64k ofs-delta shallow no-progress include-tag multi_ack_d
etailed no-done symref=HEAD:refs/heads/master agent=git/2.0.4
20:15:14.867088 pkt-line.c:46           packet:          git< 0f20ae29889d61f2e93ae00fd34f1cdb53285702
refs/heads/ab/add-interactive-show-diff-func-name
20:15:14.867094 pkt-line.c:46           packet:          git< 36dc827bc9d17f80ed4f326de21247a5d1341fbc
refs/heads/ah/doc-gitk-config
# […]
```

GIT_TRACE_PERFORMANCE 控制性能数据的日志打印。 输出显示了每个 Git 命令调用花费的时间。

```
$ GIT_TRACE_PERFORMANCE=true git gc
20:18:19.499676 trace.c:414             performance: 0.374835000 s: git command: 'git' 'pack-refs' '--a
ll' '--prune'
20:18:19.845585 trace.c:414             performance: 0.343020000 s: git command: 'git' 'reflog' 'expire
' '--all'
Counting objects: 170994, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (43413/43413), done.
Writing objects: 100% (170994/170994), done.
Total 170994 (delta 126176), reused 170524 (delta 125706)
20:18:23.567927 trace.c:414             performance: 3.715349000 s: git command: 'git' 'pack-objects' '
--keep-true-parents' '--honor-pack-keep' '--non-empty' '--all' '--reflog' '--unpack-unreachable=2.weeks
.ago' '--local' '--delta-base-offset' '.git/objects/pack/.tmp-49190-pack'
```

GIT_TRACE_SETUP 显示 Git 发现的关于版本库和交互环境的信息

```
$ GIT_TRACE_SETUP=true git status
20:19:47.086765 trace.c:315             setup: git_dir: .git
20:19:47.087184 trace.c:316             setup: worktree: /Users/ben/src/git
20:19:47.087191 trace.c:317             setup: cwd: /Users/ben/src/git
20:19:47.087194 trace.c:318             setup: prefix: (null)
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

其它

如果指定了 **GIT_SSH** ， Git 连接 SSH 主机时会用指定的程序代替 `ssh` 。 它会被用 `$GIT_SSH [username@]host [-p <port>] <command>` 的命令方式调用。 这不是配置定制 `ssh` 调用方式的最简单的方法; 它不支持额外的命令行参数， 所以你必须写一个封装脚本然后让 `GIT_SSH` 指向它。 可能用 `~/.ssh/config` 会更简单。

GIT_ASKPASS 覆盖了 `core.askpass` 配置。 这是 Git 需要向用户请求验证时用到的程序，它接受一个文本提示作为命令行参数，并在 `stdout` 中返回应答。 (查看 凭证存储_ 访问更多相关内容)

GIT_NAMESPACE 控制有命令空间的引用的访问，与 `--namespace` 标志是相同的. 这主要在服务器端有用， 如果你想在一个版本库中存储单个版本库的多个 fork, 只要保持引用是隔离的就可以。

GIT_FLUSH 强制 Git 在向标准输出增量写入时使用没有缓存的 I/O。 设置为 1 让 Git 刷新更多， 设置为 0 则使所有的输出被缓存。 默认值（若此变量未设置）是根据活动和输出模式的不同选择合适的缓存方案。

GIT_REFLOG_ACTION 让你可以指定描述性的文字写到 reflog 中。 这有个例子：

```
$ GIT_REFLOG_ACTION="my action" git commit --allow-empty -m 'my message'
[master 9e3d55a] my message
$ git reflog -1
9e3d55a HEAD@{0}: my action: my message
```

10.9 总结

总结

现在，你应该相当了解 Git 在背后都做了些什么工作，并且在一定程度上也知道了 Git 是如何实现的。 本章讨论了很多底层命令，这些命令比我们在本书其余部分学到的高层命令来得更原始，也更简洁。 从底层了解 Git 的工作原理有助于更好地理解 Git 在内部是如何运作的，也方便你能够针对特定的工作流写出自己的工具和脚本。

作为一套内容寻址文件系统，Git 不仅仅是一个版本控制系统，它同时是一个非常强大且易用的工具。 我们希望你可以借助新学到的 Git 内部原理相关知识来实现出自己的应用，并且以更高级、更得心应手的方式来驾驭 Git。

A. 其它环境中的 Git

附录A. 其它环境中的 Git

从头至尾读到了这里，你肯定已经掌握了不少使用 Git 命令行操作的知识。你学会了操作本地文件，通过网络连接你的仓库，以及与他人进行有效率的合作。但是故事并未就此结束；Git 通常只是更大的生态圈的一部分，在某些情况下使用终端并不是最合适的方式。现在就让我们来了解一下如何在其它类型的环境中更好地使用 Git，以及别的应用（包括你的）如何与 Git 进行协作。

A1.1 图形界面

图形界面

Git 的原生环境是终端。 在那里，你可以体验到最新的功能，也只有在那里，你才能尽情发挥 Git 的全部能力。 但是对于某些任务而言，纯文本并不是最佳的选择；有时候你确实需要一个可视化的展示方式，而且有些用户更习惯那种能点击的界面。

有一点请注意，不同的界面是为不同的工作流程设计的。 一些客户端的作者为了支持某种他认为高效的工作流程，经过精心挑选，只显示了 Git 功能的一个子集。 每种工具都有其特定的目的和意义，从这个角度来看，不能说某种工具比其它的“更好”。 还有请注意，没有什么事情是图形界面客户端可以做而命令行客户端不能做的；命令行始终是你可以完全操控仓库并发挥出全部力量的地方。

`gitk` 和 `git-gui`

在安装 Git 的同时，你也装好了它提供的可视化工具，`gitk` 和 `git-gui`。

`gitk` 是一个历史记录的图形化查看器。 你可以把它当作是基于 `git log` 和 `git grep` 命令的一个强大的图形操作界面。 当你需要查找过去发生的某次记录，或是可视化查看项目历史的时候，你将会用到这个工具。

使用 Gitk 的最简单方法就是从命令行打开。 只需 `cd` 到一个 Git 仓库，然后键入：

```
$ gitk [git log options]
```

Gitk 可以接受很多命令行选项，其中的大部分都直接传给底层的 `git log` 去执行了。 `--all` 可能是这其中最有用的一个, 它告诉 gitk 去尽可能地从 任何 引用查找提交并显示，而不仅仅是从 HEAD。 Gitk 的界面看起来长这样：

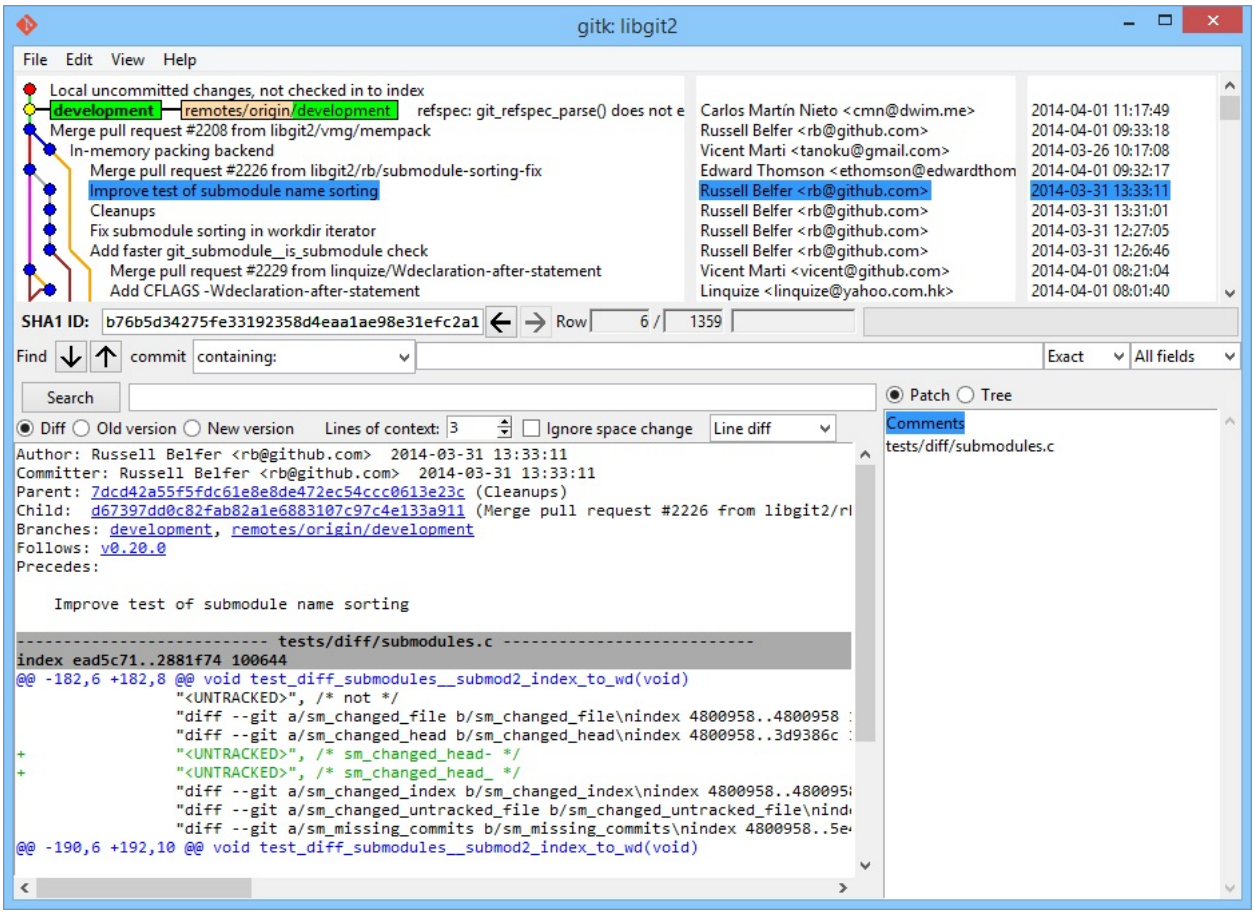


Figure 1-1. `gitk` 历史查看器。

这张图看起来就和执行 `git log --graph` 命令的输出差不多；每个点代表一次提交，线代表父子关系，而彩色的方块则用来标示一个个引用。 黄点表示 HEAD，红点表示尚未提交的本地变动。 下方的窗口用来显示当前选中的提交的具体信息；评论和补丁显示在左侧，摘要显示在右侧。 中间则是一组用来搜索历史的控件。

与之相比，`git-gui` 则主要是一个用来制作提交的工具。 打开它的最简单方法也是从命令行启动：

```
$ git gui
```

它的界面长这个样子：

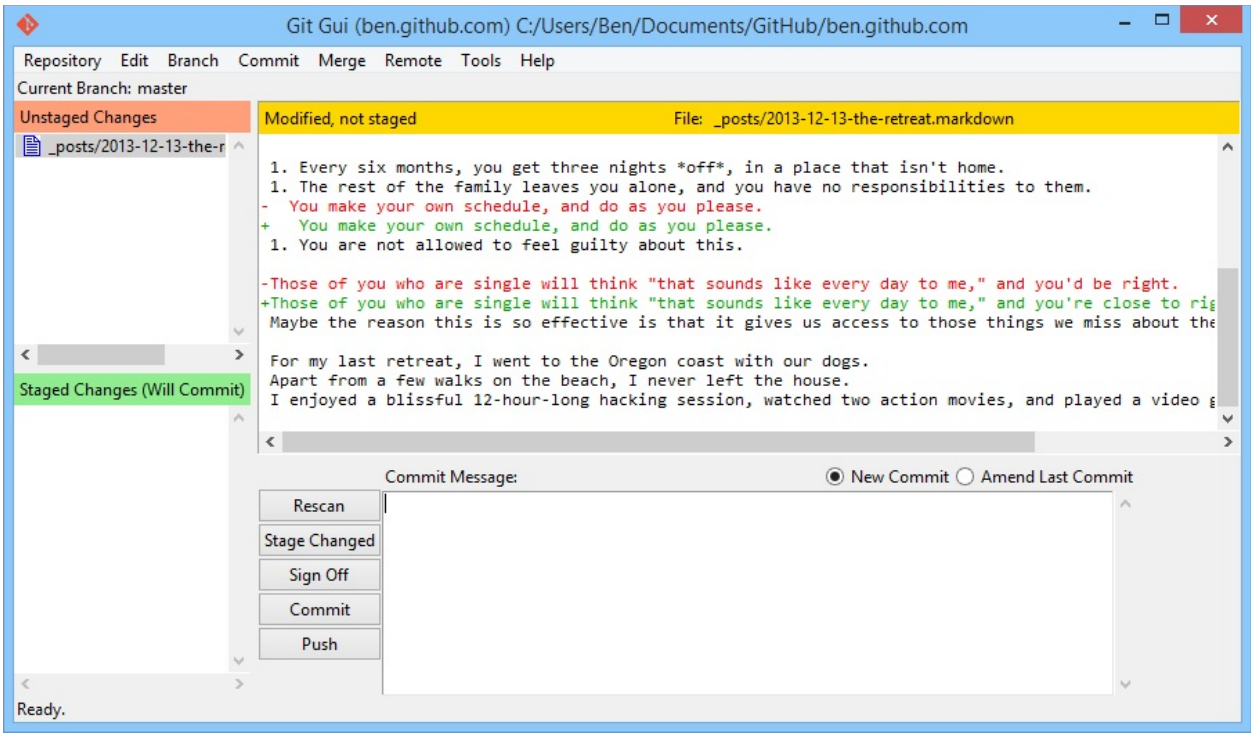


Figure 1-2. `git-gui` 提交工具。

左侧是索引区；未暂存的修改显示在上方，已暂存的修改显示在下方。 你可以通过点击文件名左侧的图标来将该文件在暂存状态与未暂存状态之间切换，你也可以通过选中一个文件名来查看它的详情。

右侧窗口的上方以 diff 格式来显示当前选中文件发生了变动的地方。 你可以通过右击某一区块或行从而将这一区块或行放入暂存区。

右侧窗口的下方是写日志和执行操作的地方。 在文本框中键入日志然后点击 “提交” 就和执行 `git commit` 的效果差不多。 如果你想要修订上一次提交, 可以选中“修订” 按钮，上次一提交的内容就会显示在 “暂存区”。 然后你就可以简单的对修改进行暂存和取消暂存操作，更新提交日志，然后再次点击 “提交” 用这个新的提交来覆盖上一次提交。

`gitk` 和 `git-gui` 就是针对某种任务设计的工具的两个例子。 它们分别为了不同的目的（即查看历史和制作提交）而进行了精简，略去了用不到的功能。

Mac 和 Windows 上的 GitHub 客户端

GitHub 发布了两个面向工作流程的 Git 客户端：Windows 版，和 Mac 版。 它们很好的展示了一个面向工作流程的工具应该是什么样子——专注于提升那些常用的功能及其协作的可用性，而不是实现 Git 的 所有 功能. 它们看起来长这个样子：

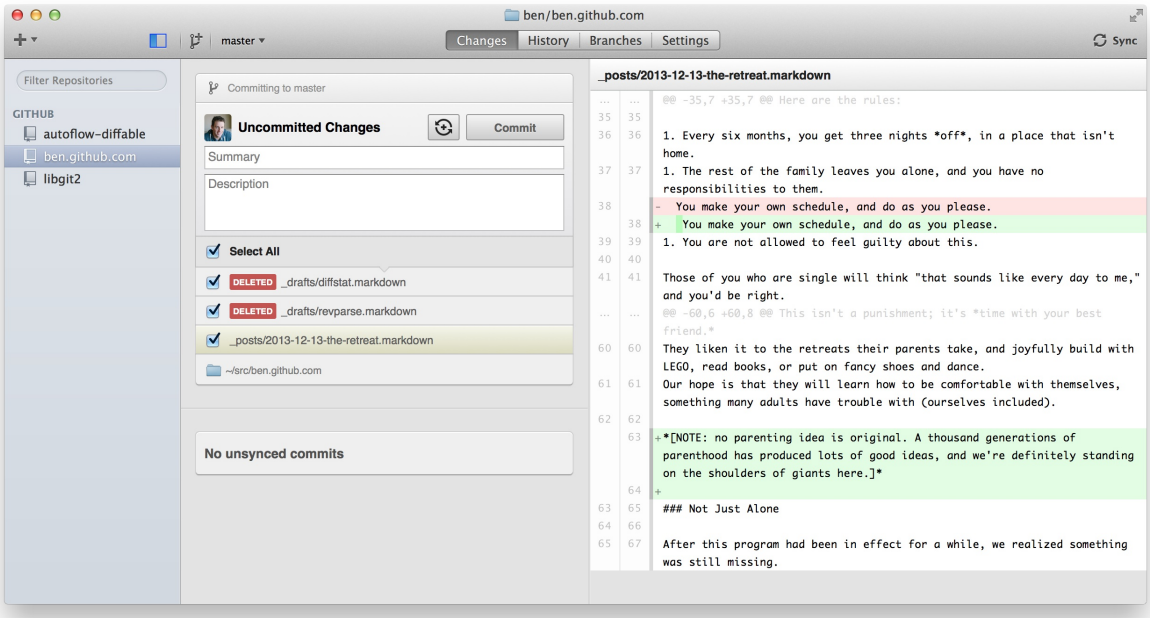


Figure 1-3. GitHub Mac 客户端。

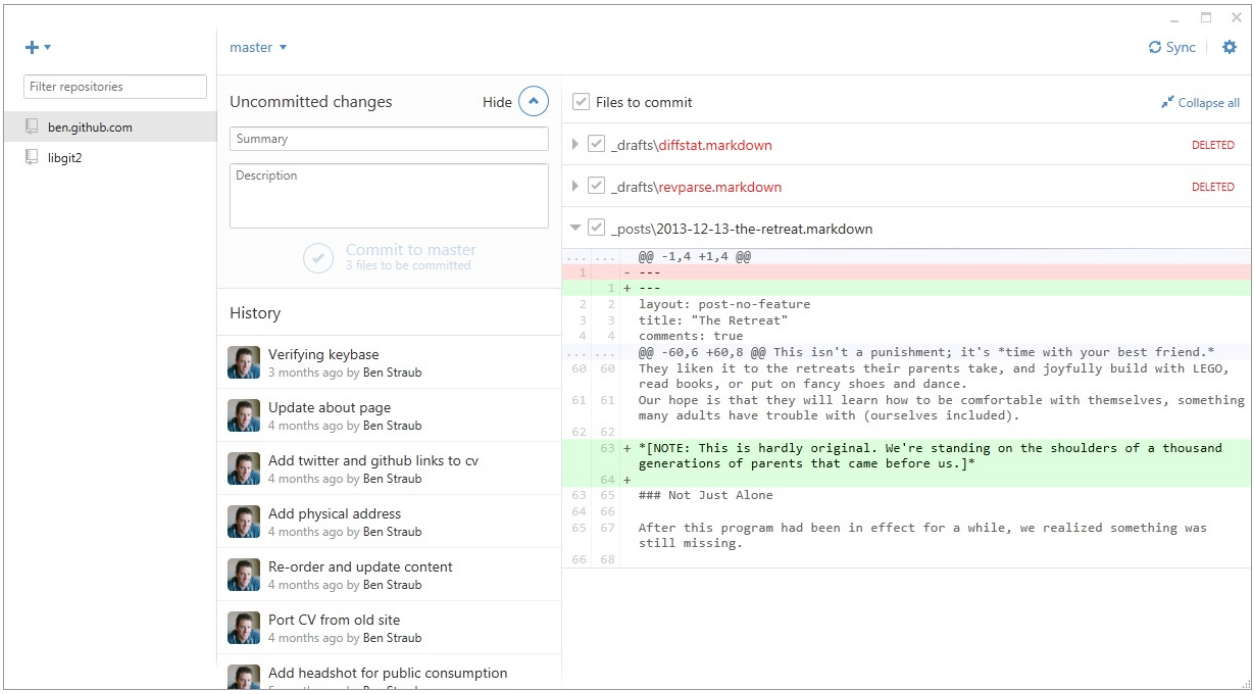


Figure 1-4. GitHub Windows 客户端。

我们在设计的时候就努力将二者的外观和操作体验都保持一致，因此本章会把他们当做同一个产品来介绍。 我们并不会详细地介绍该工具的每一个功能（因为它们本身也有文档），但请快速了解一下 “变更” 窗口（你大部分时间都会花在使用该窗口上）的以下几点：

- 左侧是正在追踪的仓库的列表；通过点击左上方的 “+” 图标，你可以添加一个需要追踪的仓库（既可以是通过 clone，也可以从本地添加）。
- 中间是输入-提交区，你可以在这里输入提交日志，以及选择哪些文件需要被提交。（在 Windows 上，提交历史就显示在这个区域的下方；在 Mac 上，提交历史有一个单独的窗口）
- 右侧是修改查看区，它会告诉你工作目录里哪些东西被修改了（译注：修改模式），或选中的提交里包括了哪些修改（译注：历史模式）。
- 最后需要熟悉的是右上角的 “Sync” 按钮，你主要通过这个按钮来进行网络上的交互。

NOTE

你不需要注册 GitHub 账号也可以使用这些工具。 尽管它们是按照 GitHub 推荐的工作流程来设计的，并突出提升了一些 GitHub 的服务体验，但它们可以在任何 Git 仓库上工作良好，也可以通过网络连接到任意 Git 主机。

安装

GitHub 的 Windows 客户端可以从 https://windows.github.com 下载，Mac 客户端可以从 https://mac.github.com下载。 第一次打开软件时，它会引导你进行一系列的首次使用设置，例如设置你的姓名和电子邮件，它还会智能地帮你调整一些常用的默认设置，例如凭证缓存和 CRLF 的处理方式。

它们都是“绿色软件”——如果软件打开发现有更新，下载和安装升级包都是在后台完成的。 为方便起见它们还打包了一份 Git，也就是说你一旦安装好就再也无需劳心升级的事情了。 Windows 的客户端还提供了快捷方式，可以启动装了 Posh-git 插件的 Powershell，在本章的后面一节我们会详细介绍这方面的内容。

接下来我们给它设置一些工作仓库。 客户端会显示你在 GitHub 上有权限操作的仓库的列表，你可以选择一个然后一键克隆。 如果你本地已经建立了仓库，只需要用鼠标把它从 Finder 或 Windows 资源管理器拖进 GitHub 客户端窗口，就可以把该仓库添加到左侧的仓库列表里面去了。

推荐的工作流程

安装并配置好以后，你就可以使用 GitHub 客户端来执行一些常见的 Git 任务。 该工具所推荐的工作流程有时也被叫做 “GitHub 流”。 我们在 GitHub 流程 一节中对此有详细的介绍，其要点是 (a) 你会提交到一个分支；(b) 你需要经常与远程仓库保持同步。

两个平台上的客户端在分支管理上有所不同。 在 Mac 上，创建分支的按钮在窗口的上方：

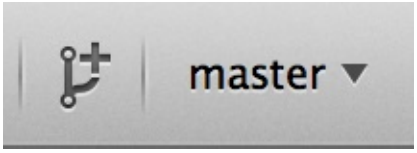


Figure 1-5. Mac 上的“创建分支”按钮。

在 Windows 上，你可以通过在分支切换挂件中输入新分支的名称来完成创建：

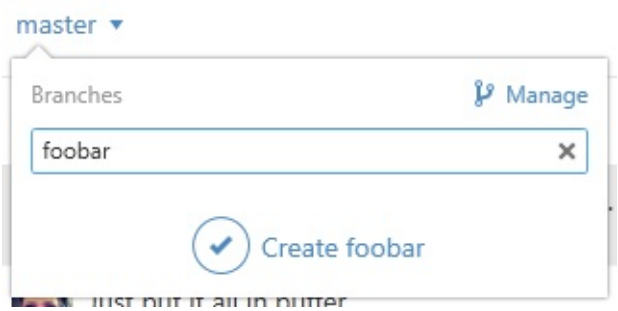


Figure 1-6. 在 Windows 上创建分支。

分支创建好以后，新建提交就变得非常简单直接了。 现在工作目录中做一些修改，然后切换到 GitHub 客户端窗口，你所做的修改就会显示在那里。 输入提交日志，选中那些需要被包含在本次提交中的文件，然后点击 “提交” 按钮（也可以在键盘上按 ctrl-enter 或 ⌘-enter）。

“同步” 功能是你在网络上和其它仓库交互的主要途径。 push，fetch，merge，和 rebase 在 Git 内部是一连串独立的操作, 而 GitHub 客户端将这些操作都合并成了单独一个功能。 你点击同步按钮时实际上会发生如下这些操作：

1. `git pull --rebase`。 如果上述命令由于存在合并冲突而失败，则会退而执行 `git pull --no-rebase`。
2. `git push`。

如果你遵循推荐的工作流程，以上就是最常用的一系列命令，因此将它们合并为一个让事情简单了很多。

小结
这些工具是为其各自针对的工作流程所量身定做的。 开发者和非开发者可以轻松地在分分钟内就搭建起项目协作环境，它们还内置了其它辅助最佳实践的功能。 但是，如果你的工作流程有所不同，或者你需要在进行网络操作时有更多的控制，那么建议你考虑一下其它客户端或者使用命令行。

其它图形界面

除此之外，还有许许多多其它的图形化 Git 客户端，其中既有单一功能的定制工具，也有试图提供 Git 所有功能的复杂应用。 Git 的官方网站整理了一份时下最流行的客户端的清单 http://git-scm.com/downloads/guis。 在 Git 的维基站点还可以看到一份更全的清单https://git.wiki.kernel.org/index.php/Interfaces,_frontends,_and_tools#Graphical_Interfaces.

A1.2 Visual Studio 中的 Git

Visual Studio 中的 Git

从 Visual Studio 2013 Update 1 版本开始，Visual Studio 用户可以在他们的 IDE 中直接使用内嵌的 Git 客户端。 Visual Studio 集成源代码版本控制特性已经有很长一段时间，但面向的是集中式、文件锁定方式的系统，Git 并不能很好地符合这种工作流程。 Visual Studio 2013 中已经支持 Git，并独立于原有版本管理系统，这使得 Visual Studio 和 Git 能更好地相互适应。

想要找到这个特性，在 Visual Studio 中打开一个已经用 Git 管理的项目（或者直接在项目目录中 `git init` ），选择菜单 View > Team Explorer。 你将看到 "Connect" 视图，大概如下图所示：

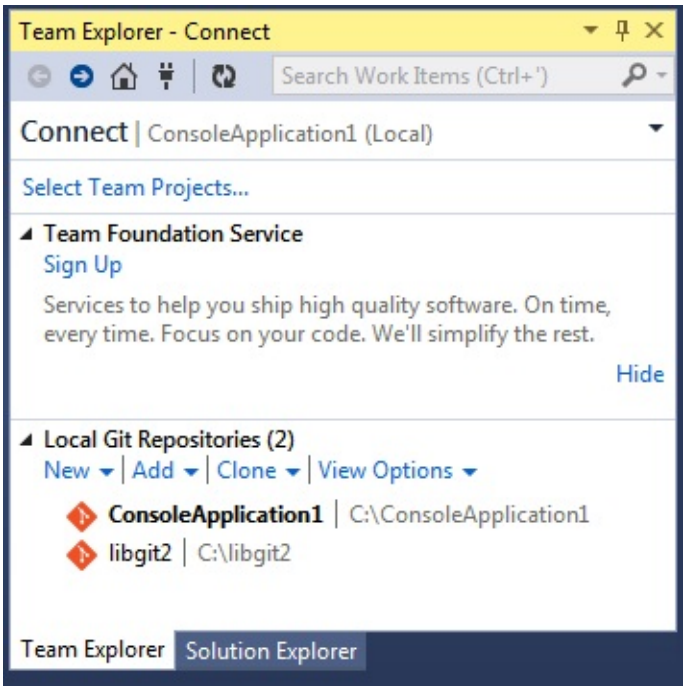


Figure 1-7. 从 Team Explorer 中连接 Git 仓库。

Visual Studio 能够记住所有你打开过的用 Git 管理的项目，它们都在下方的列表中。 如果没看到你想要的项目，点击 "Add" 按钮，添加项目工作目录的路径。 双击其中一个本地的 Git 仓库会将你带入 "Home" 视图，大概如 Figure 1-8 所示。 这是一个执行 Git 操作的操作中心；当你 正在编写 代码的时候，你可能主要关注 "Changes" 视图，当需要拉取同伴的改动时，你将使用 "Unsynced Commits" 和 "Branches" 视图。

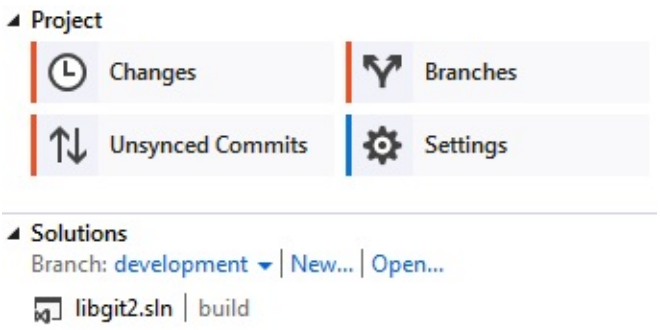


Figure 1-8. Visual Studio 中的 Git 仓库的 “Home” 视图。

Visual Studio 现在拥有一套着眼于任务的强大 Git 操作界面。 它包括线性的历史视图、diff 视图、远程仓库操作命令，以及其它很多功能。 这个特性的完整文档（放在这里并不合适）请参阅http://msdn.microsoft.com/en-us/library/hh850437.aspx 。

A1.3 Eclipse 中的 Git

Eclipse 中的 Git

Eclipse 附带了一个名为 Egit 的插件，它提供了一个非常完善的 Git 操作接口。 这个插件可以通过切换到 Git 视图来使用：(Window > Open Perspective > Other... ， 然后选择 “Git” ）。

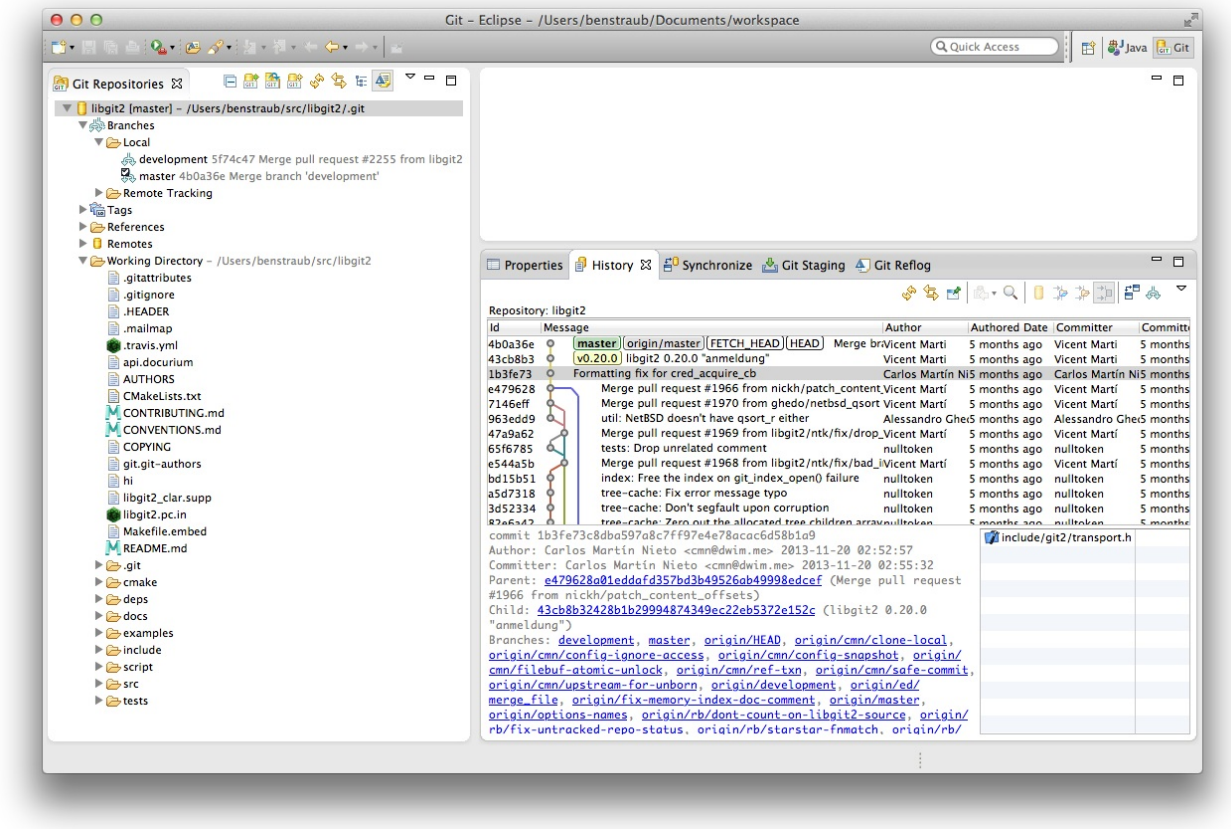


Figure 1-9. Eclipse 中 EGit 的界面环境。

EGit 提供了许多强大的帮助文档，你能通过下面的操作来访问它：单击菜单 Help > Help Contents，然后从内容列表中选择 “EGit Documentation” 节点。

A1.4 Bash 中的 Git

Bash 中的 Git

如果你是一名 Bash 用户，你可以从中发掘出一些 Shell 的特性，让你在使用 Git 时更加随心所欲。 实际上 Git 附带了几个 Shell 的插件，但是这些插件并不是默认打开的。

首先，你需要从 Git 源代码中获得一份 `contrib/completion/git-completion.bash` 文件的拷贝。 将这个文件复制到一个相对便捷的目录，例如你的 Home 目录，并且将它的路径添加到 `.bashrc` 中：

```
. ~/git-completion.bash
```

做完这些之后，请将你当前的目录切换到某一个 Git 仓库，并且输入：

```
$ git chec<tab>
```

……此时 Bash 将会把上面的命令自动补全为 `git checkout`。 在适当的情况下，这项功能适用于 Git 所有的子命令、命令行参数、以及远程仓库与引用名。

这项功能也可以用于你自己定义的提示符（ `prompt` ），显示当前目录下 Git 仓库的信息。 根据你的需要，这个信息可以简单或复杂，这里通常有大多数人想要的几个关键信息，比如当前分支信息和当前工作目录的状态信息。 要添加你自己的提示符（ `prompt` ），只需从 Git 源版本库复制 `contrib/completion/git-prompt.sh` 文件到你的 Home 目录(或其他便于你访问与管理的目录)， 并在 `.bashrc` 里添加这个文件路径，类似于下面这样：

```
. ~/git-prompt.sh
```

`\w` 表示打印当前工作目录，`\$` 打印 `$` 部分的提示符（ `prompt` ），`__git_ps1 " (%s)"` 表示通过格式化参数符（ `%s` ）调用 `git-prompt.sh` 脚本中提供的函数。 因为有了这个自定义提示符，现在你的 `Bash` 提示符（ `prompt` ）在 Git 仓库的任何子目录中都将显示成这样：



Figure 1-10. 自定义的 `bash` 提示符（ `prompt` ）.

这两个脚本都提供了很有帮助的文档；浏览 `git-completion.bash` 和 `git-prompt.sh` 的内容以获得更多信息。

A1.5 Zsh 中的 Git

Zsh 中的 Git

Git 还为 Zsh 提供了一个 Tab 补全库。 复制 `contrib/completion/git-completion.zsh` 到你的 home 目录，然后在 `.zshrc` 中 source 即可。 相对于 Bash，Zsh 的接口更加强大：

```
$ git che<Tab>
check-attr        -- 显示 gitattributes 信息
check-ref-format  -- 检查引用名称是否符合规范
checkout          -- 从工作区中检出分支或路径
checkout-index    -- 从暂存区拷贝文件至工作目录
cherry            -- 查找没有被合并至上游的提交
cherry-pick       -- 从一些已存在的提交中应用更改
```

意义不明的 Tab 补全并不仅仅会被列出；它们还会有帮助性的描述，你可以通过不断敲击 Tab 以图形方式浏览补全列表。 该功能可用于 Git 命令、它们的参数和在仓库中内容的名称（例如 refs 和 remotes），还有文件名和其他所有 Zsh 知道如何去补全的项目。

在提示符自定义方面，Zsh 很好地兼容了Bash，并允许你同时使用一个右侧提示符。 把如下代码添加至你的 `~/.zshrc` 文件中，就可以在右侧显示分支名称：

```
setopt prompt_subst
```

当你的命令行位于一个 Git 仓库目录时，在任何时候，都可以在命令行窗口右侧显示当前分支。 它看起来像这样：

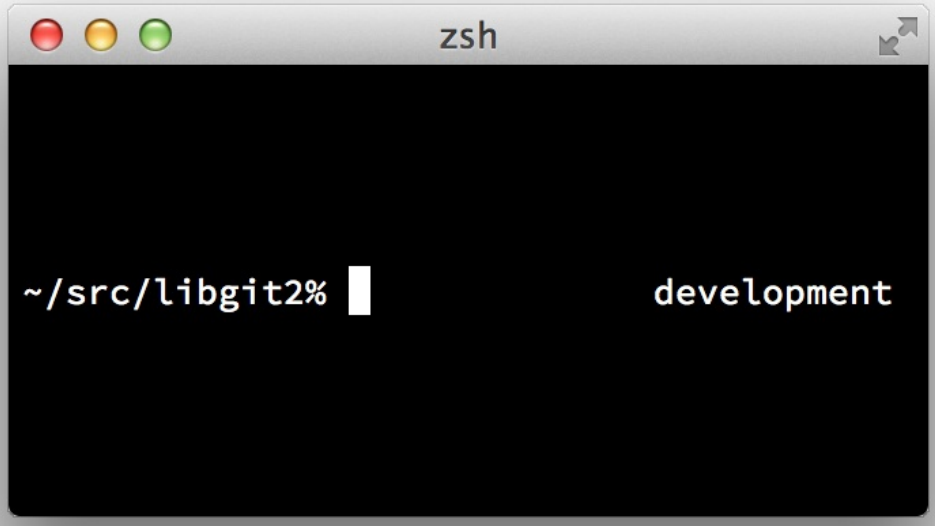


Figure 1-11. 自定义 `zsh` 提示符.

Zsh 本身已足够强大，但还有一些专门为它打造的完整框架，使它更加完善。 其中之一名为 "oh-my-zsh"，你可以在 https://github.com/robbyrussell/oh-my-zsh 找到它。 oh-my-zsh 的扩展系统包含强大的 Git Tab 补全功能，且许多提示符 "主题" 可以展示版本控制数据。 Figure 1-12 只是可以其中一个可以通过该系统实现的例子。

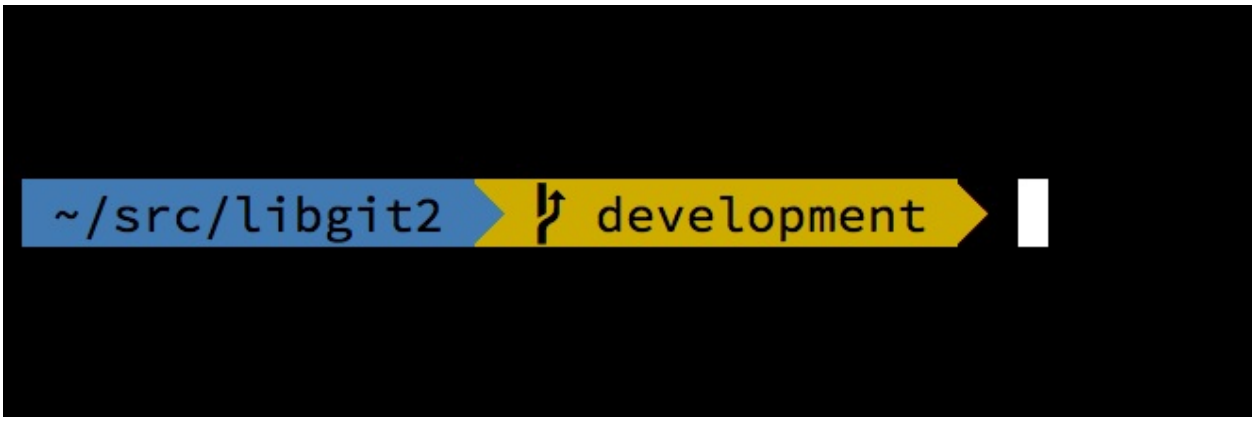


Figure 1-12. 一个 oh-my-zsh 主题的示例.

A1.6 Powershell 中的 Git

Powershell 中的 Git

Windows 中的普通命令行终端（`cmd.exe`）无法自定义 Git 使用体验，但是如果你正在使用 Powershell，那么你就十分幸运了。 一个名为 Posh-Git (https://github.com/dahlbyk/posh-git) 的扩展包提供了强大的 tab 补全功能， 并针对提示符进行了增强，以帮助你聚焦于你的仓库状态。 它看起来像：

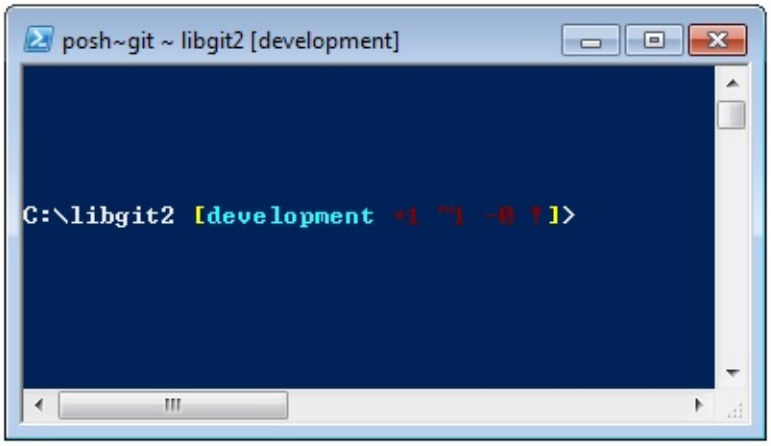


Figure 1-13. 附带了 Posh-Git 扩展包的 Powershell。

如果你已经在 Windows 上安装了 GitHub，Posh-Git 也会被安装，你只需要添加以下两行到你的 `profile.ps1` 文件 (文件位于 `C:\Users\<username>\Documents\WindowsPowerShell`):

```
. (Resolve-Path "$env:LOCALAPPDATA\GitHub\shell.ps1")
. $env:github_posh_git\profile.example.ps1
```

如果你没有在 Windows 上安装 GitHub，只需要从 (https://github.com/dahlbyk/posh-git) 下载一份 Posh-Git 发行版，并且解压至 `WindowsPowershell` 目录。 然后以管理员权限打开 Powershell 提示符，并且执行下面的命令：

```
> Set-ExecutionPolicy RemoteSigned -Scope CurrentUser -Confirm
> cd ~\Documents\WindowsPowerShell\posh-git
> .\install.ps1
```

它将会向你的 `profile.ps1` 文件添加适当的内容，Posh-Git 将会在下次打开提示符时被启用。

A1.7 总结

总结

你已经学会了如何从日常工具中发挥 Git 的强大力量，以及从自己的程序中访问 Git 仓库的方法。

B. 将 Git 嵌入你的应用

附录B. 将 Git 嵌入你的应用

假设你的应用程序的目标人群是开发者，如果它能够被整合进一些源码控制的功能，那真真是极好的。甚至对于一个例如文档编辑器之类的不是为开发者而设计的应用程序，它们也可能从版本控制系统中受益，并且 Git 的实现方式在很多情况下都表现得非常出色。

如果你想将 Git 整合进你的应用程序的话，一般来说你有三种可能的选择：启动一个 shell 来使用 Git 的命令行工具；使用 Libgit2；或者使用 JGit。

A2.1 命令行 Git 方式

命令行 Git 方式

一种方式就是启动一个 shell 进程并在里面使用 Git 的命令行工具来完成任务。 这种方式看起来很循规蹈矩，但是它的优点也因此而来，就是支持所有的 Git 的特性。 它也碰巧相当简单，因为几乎所有运行时环境都有一个相对简单的方式来调用一个带有命令行参数的进程。 然而，这种方式也有一些固有的缺点。

一个就是所有的输出都是纯文本格式。 这意味着你将被迫解析 Git 的有时会改变的输出格式，以随时了解它工作的进度和结果。更糟糕的是，这可能是无效率并且容易出错的。

另外一个就是令人捉急的错误修复能力。 如果一个版本库被莫名其妙地损毁，或者用户使用了一个奇奇怪怪的配置， Git 只会简单地拒绝表现自己的强大能力。

还有一个就是进程的管理。 Git 会要求你在一个独立的进程中维护一个 shell 环境，这可能会无谓地增加复杂性。 试图协调许许多多的类似的进程（尤其是在某些情况下，当不同的进程在访问相同的版本库时）是对你的能力的极大的挑战。

A2.2 Libgit2

Libgit2

另外一种可以供你使用的是 Libgit2。 Libgit2 是一个 Git 的非依赖性的工具，它致力于为其他程序使用 Git 提供更好的 API。 你可以在 http://libgit2.github.com 找到它。

首先，让我们来看一下 C API 长啥样。 这是一个旋风式旅行。

```
// 打开一个版本库
git_repository *repo;
int error = git_repository_open(&repo, "/path/to/repository");

// 逆向引用 HEAD 到一个提交
git_object *head_commit;
error = git_revparse_single(&head_commit, repo, "HEAD^{commit}");
git_commit *commit = (git_commit*)head_commit;

// 显示这个提交的一些详情
printf("%s", git_commit_message(commit));
const git_signature *author = git_commit_author(commit);
printf("%s <%s>\n", author->name, author->email);
const git_oid *tree_id = git_commit_tree_id(commit);

// 清理现场
git_commit_free(commit);
git_repository_free(repo);
```

前两行打开一个 Git 版本库。 这个 `git_repository` 类型代表了一个在内存中带有缓存的指向一个版本库的句柄。 这是最简单的方法，只是你必须知道一个版本库的工作目录或者一个 `.git` 文件夹的精确路径。 另外还有 `git_repository_open_ext` ，它包括了带选项的搜索， `git_clone` 及其同类可以用来做远程版本库的本地克隆， `git_repository_init` 则可以创建一个全新的版本库。

第二段代码使用了一种 rev-parse 语法（要了解更多，请看 分支引用 ）来得到 HEAD 真正指向的提交。返回类型是一个 `git_object` 指针，它指代位于版本库里的 Git 对象数据库中的某个东西。 `git_object` 实际上是几种不同的对象的 “父” 类型，每个 “子” 类型的内存布局和 `git_object` 是一样的，所以你能安全地把它们转换为正确的类型。 在上面的例子中， `git_object_type(commit)` 会返回 `GIT_OBJ_COMMIT` ，所以转换成 `git_commit` 指针是安全的。

下一段展示了如何访问一个提交的详情。 最后一行使用了 `git_oid` 类型，这是 Libgit2 用来表示一个 SHA-1 哈希的方法。

从这个例子中，我们可以看到一些模式：

- 如果你声明了一个指针，并在一个 Libgit2 调用中传递一个引用，那么这个调用可能返回一个 int 类型的错误码。 值 `0` 表示成功，比它小的则是一个错误。

- 如果 Libgit2 为你填入一个指针，那么你有责任释放它。
- 如果 Libgit2 在一个调用中返回一个 `const` 指针，你不需要释放它，但是当它所指向的对象被释放时它将不可用。
- 用 C 来写有点蛋疼。

最后一点意味着你应该不会在使用 Libgit2 时编写 C 语言程序。 但幸运的是，有许多可用的各种语言的绑定，能让你在特定的语言和环境中更加容易的操作 Git 版本库。 我们来看一下下面这个用 Libgit2 的 Ruby 绑定写成的例子，它叫 Rugged，你可以在https://github.com/libgit2/rugged 找到它。

```
repo = Rugged::Repository.new('path/to/repository')
commit = repo.head.target
puts commit.message
puts "#{commit.author[:name]} <#{commit.author[:email]}>"
tree = commit.tree
```

你可以发现，代码看起来更加清晰了。 首先， Rugged 使用异常机制，它可以抛出类似于 `ConfigError` 或者 `ObjectError` 之类的东西来告知错误的情况。 其次，不需要明确资源释放，因为 Ruby 是支持垃圾回收的。 我们来看一个稍微复杂一点的例子：从头开始制作一个提交。

```
blob_id = repo.write("Blob contents", :blob)

index = repo.index
index.read_tree(repo.head.target.tree)
index.add(:path => 'newfile.txt', :oid => blob_id)

sig = {
    :email => "bob@example.com",
    :name => "Bob User",
    :time => Time.now,
}

commit_id = Rugged::Commit.create(repo,
    :tree => index.write_tree(repo),
    :author => sig,
    :committer => sig,
    :message => "Add newfile.txt",
    :parents => repo.empty? ? [] : [ repo.head.target ].compact,
    :update_ref => 'HEAD',
)
commit = repo.lookup(commit_id)
```



创建一个新的 blob ，它包含了一个新文件的内容。



将 HEAD 提交树填入索引，并在路径 `newfile.txt` 增加新文件。

这就在 ODB 中创建了一个新的树，并在一个新的提交中使用它。

我们在 author 栏和 committer 栏使用相同的签名。

提交的信息。

当创建一个提交时，你必须指定这个新提交的父提交。 这里使用了 HEAD 的末尾作为单一的父提交。

在做一个提交的过程中， Rugged （和 Libgit2 ）能在需要时更新引用。

返回值是一个新提交对象的 SHA-1 哈希，你可以用它来获得一个 `Commit` 对象。

Ruby 的代码很好很简洁，另一方面因为 Libgit2 做了大量工作，所以代码运行起来其实速度也不赖。 如果你不是一个 Ruby 程序员，我们在 其它绑定 有提到其它的一些绑定。

高级功能

Libgit2 有几个超过核心 Git 的能力。 例如它的可定制性：Libgit2 允许你为一些不同类型的操作自定义的“后端” ，让你得以使用与原生 Git 不同的方式存储东西。 Libgit2 允许为自定义后端指定配置、引用的存储以及对象数据库，

我们来看一下它究竟是怎么工作的。 下面的例子借用自 Libgit2 团队提供的后端样本集 （可以在https://github.com/libgit2/libgit2-backends 上找到）。 一个对象数据库的自定义后端是这样建立的：

```
git_odb *odb;
int error = git_odb_new(&odb);

git_odb_backend *my_backend;
error = git_odb_backend_mine(&my_backend, /*…*/);

error = git_odb_add_backend(odb, my_backend, 1);

git_repository *repo;
error = git_repository_open(&repo, "some-path");
error = git_repository_set_odb(odb);
```

(注意：这个错误被捕获了，但是没有被处理。我们希望你的代码比我们的更好。)

初始化一个空的对象数据库（ ODB ） “前端” ，它将被作为一个用来做真正的工作的 “后端” 的容器。

初始化一个自定义 ODB 后端。

为这个前端增加一个后端。

打开一个版本库，并让它使用我们的 ODB 来寻找对象。

但是 `git_odb_backend_mine` 是个什么东西呢？ 嗯，那是一个你自己的 ODB 实现的构造器，并且你能在那里做任何你想做的事，前提是你能正确地填写 `git_odb_backend` 结构。 它看起来应该是这样的：

```
typedef struct {
    git_odb_backend parent;

    // 其它的一些东西
    void *custom_context;
} my_backend_struct;

int git_odb_backend_mine(git_odb_backend **backend_out, /*…*/)
{
    my_backend_struct *backend;

    backend = calloc(1, sizeof (my_backend_struct));

    backend->custom_context = …;

    backend->parent.read = &my_backend__read;
    backend->parent.read_prefix = &my_backend__read_prefix;
    backend->parent.read_header = &my_backend__read_header;
    // ……

    *backend_out = (git_odb_backend *) backend;

    return GIT_SUCCESS;
}
```

`my_backend_struct` 的第一个成员必须是一个 `git_odb_backend` 结构，这是一个微妙的限制：这样就能确保内存布局是 Libgit2 的代码所期望的样子。 其余都是随意的，这个结构的大小可以随心所欲。

这个初始化函数为该结构分配内存，设置自定义的上下文，然后填写它支持的 `parent` 结构的成员。 阅

读 Libgit2 的 `include/git2/sys/odb_backend.h` 源码以了解全部调用签名，你特定的使用环境会帮你决定使用哪一种调用签名。

其它绑定

Libgit2 有很多种语言的绑定。 在这篇文章中，我们展现了一个使用了几个更加完整的绑定包的小例子，这些库存在于许多种语言中，包括 C++、Go、Node.js、Erlang 以及 JVM ，它们的成熟度各不相同。 官方的绑定集合可以通过浏览这个版本库得到：https://github.com/libgit2[] 。 我们写的代码将返回当前 HEAD 指向的提交的提交信息(就像 `git log -1` 那样)。

LibGit2Sharp

如果你在编写一个 .NET 或者 Mono 应用，那么 LibGit2Sharp (https://github.com/libgit2/libgit2sharp) 就是你所需要的。 这个绑定是用 C# 写成的，并且已经采取许多措施来用令人感到自然的 CLR API 包装原始的 Libgit2 的调用。 我们的例子看起来就像这样：

```
new Repository(@"C:\path\to\repo").Head.Tip.Message;
```

对于 Windows 桌面应用，一个叫做 NuGet 的包会让你快速上手。

objective-git

如果你的应用运行在一个 Apple 平台上，你很有可能使用 Objective-C 作为实现语言。 Objective-Git (https://github.com/libgit2/objective-git) 是这个环境下的 Libgit2 绑定。 一个例子看起来类似这样：

```
GTRepository *repo =
    [[GTRepository alloc] initWithURL:[NSURL fileURLWithPath: @"/path/to/repo"] error:NULL];
NSString *msg = [[[repo headReferenceWithError:NULL] resolvedTarget] message];
```

Objective-git 与 Swift 完美兼容，所以你把 Objective-C 落在一边的时候不用恐惧。

pygit2

Python 的 Libgit2 绑定叫做 Pygit2 ，你可以在 http://www.pygit2.org/ 找到它。 我们的示例程序：

```
pygit2.Repository("/path/to/repo") # 打开版本库
    .head                          # get the current branch
    .peel(pygit2.Commit)           # walk down to the commit
    .message                       # read the message
```

扩展阅读

当然，完全阐述 Libgit2 的能力已超出本书范围。 如果你想了解更多关于 Libgit2 的信息，可以浏览它的 API 文档： https://libgit2.github.com/libgit2, 以及一系列的指南：https://libgit2.github.com/docs. 对于其它的绑定，检查附带的 README 和测试文件，那里通常有简易教程，以及指向拓展阅读的链接。

A2.3 JGit

JGit

如果你想在一个 Java 程序中使用 Git ，有一个功能齐全的 Git 库，那就是 JGit 。 JGit 是一个用 Java 写成的功能相对健全的 Git 的实现，它在 Java 社区中被广泛使用。 JGit 项目由 Eclipse 维护，它的主页在 http://www.eclipse.org/jgit 。

起步

有很多种方式可以让 JGit 连接你的项目，并依靠它去写代码。 最简单的方式也许就是使用 Maven 。你可以通过在你的 pom.xml 文件里的 `<dependencies>` 标签中增加像下面这样的片段来完成这个整合。

```
<dependency>
    <groupId>org.eclipse.jgit</groupId>
    <artifactId>org.eclipse.jgit</artifactId>
    <version>3.5.0.201409260305-r</version>
</dependency>
```

在你读到这段文字时 `version` 很可能已经更新了，所以请浏览 http://mvnrepository.com/artifact/org.eclipse.jgit/org.eclipse.jgit 以获取最新的仓库信息。 当这一步完成之后， Maven 就会自动获取并使用你所需要的 JGit 库。

如果你想自己管理二进制的依赖包，那么你可以从 http://www.eclipse.org/jgit/download 获得预构建的 JGit 二进制文件。 你可以像下面这样执行一个命令来将它们构建进你的项目。

```
javac -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App.java
```

底层命令

JGit 的 API 有两种基本的层次：底层命令和高层命令。 这个两个术语都来自 Git ，并且 JGit 也被按照相同的方式粗略地划分：高层 API 是一个面向普通用户级别功能的友好的前端（一系列普通用户使用 Git 命令行工具时可能用到的东西），底层 API 则直接作用于低级的仓库对象。

大多数 JGit 会话会以 `Repository` 类作为起点，你首先要做的事就是创建一个它的实例。 对于一个基于文件系统的仓库来说（嗯， JGit 允许其它的存储模型），用 `FileRepositoryBuilder` 完成它。

```
// 创建一个新仓库
Repository newlyCreatedRepo = FileRepositoryBuilder.create(
    new File("/tmp/new_repo/.git"));
newlyCreatedRepo.create();

// 打开一个存在的仓库
```

```
Repository existingRepo = new FileRepositoryBuilder()
    .setGitDir(new File("my_repo/.git"))
    .build();
```

无论你的程序是否知道仓库的确切位置，builder 中的那个流畅的 API 都可以提供给它寻找仓库所需所有信息。 它可以使用环境变量（ `.readEnvironment()` ），从工作目录的某处开始并搜索（ `.setWorkTree(…).findGitDir()` ），或者仅仅只是像上面那样打开一个已知的 `.git` 目录。

当你拥有一个 `Repository` 实例后，你就能对它做各种各样的事。 下面是一个速览：

```
// 获取引用
Ref master = repo.getRef("master");

// 获取该引用所指向的对象
ObjectId masterTip = master.getObjectId();

// Rev-parse
ObjectId obj = repo.resolve("HEAD^{tree}");

// 装载对象原始内容
ObjectLoader loader = repo.open(masterTip);
loader.copyTo(System.out);

// 创建分支
RefUpdate createBranch1 = repo.updateRef("refs/heads/branch1");
createBranch1.setNewObjectId(masterTip);
createBranch1.update();

// 删除分支
RefUpdate deleteBranch1 = repo.updateRef("refs/heads/branch1");
deleteBranch1.setForceUpdate(true);
deleteBranch1.delete();

// 配置
Config cfg = repo.getConfig();
String name = cfg.getString("user", null, "name");
```

这里完成了一大堆事情，所以我们还是一次理解一段的好。

第一行获取一个指向 `master` 引用的指针。 JGit 自动抓取位于 `refs/heads/master` 的 真正的 master 引用，并返回一个允许你获取该引用的信息的对象。 你可以获取它的名字（ `.getName()` ），或者一个直接引用的目标对象（ `.getObjectId()` ），或者一个指向该引用的符号指针（ `.getTarget()` ）。 引用对象也经常被用来表示标签的引用和对象，所以你可以询问某个标签是否被“削除” 了，或者说它指向一个标签对象的（也许很长的）字符串的最终目标。

第二行获得以 `master` 引用的目标，它返回一个 ObjectId 实例。 不管是否存在于一个 Git 对象的数据库，ObjectId 都会代表一个对象的 SHA-1 哈希。 第三行与此相似，但是它展示了 JGit 如何处理 rev-parse 语法（要了解更多，请看 分支引用 ），你可以传入任何 Git 了解的对象说明符，然后 JGit 会返回该对象的一个有效的 ObjectId ，或者 `null` 。

接下来两行展示了如何装载一个对象的原始内容。 在这个例子中，我们调用

`ObjectLoader.copyTo()` 直接向标准输出流输出对象的内容，除此之外 ObjectLoader 还带有读取对象的类型和长度并将它以字节数组返回的方法。 对于一个（ `.isLarge()` 返回 `true` 的）大的对象，你可以调用 `.openStream()` 来获得一个类似 InputStream 的对象，它可以在没有一次性将所有数据拉到内存的前提下读取对象的原始数据。

接下来几行展现了如何创建一个新的分支。 我们创建一个 RefUpdate 实例，配置一些参数，然后调用 `.update()` 来确认这个更改。 删除相同分支的代码就在这行下面。 记住必须先 `.setForceUpdate(true)` 才能让它工作，否则调用 `.delete()` 只会返回 `REJECTED` ，然后什么都没有发生。

最后一个例子展示了如何从 Git 配置文件中获取 `user.name` 的值。 这个 Config 实例使用我们先前打开的仓库做本地配置，但是它也会自动地检测并读取全局和系统的配置文件。

这只是底层 API 的冰山一角，另外还有许多可以使用的方法和类。 还有一个没有放在这里说明的，就是 JGit 是用异常机制来处理错误的。 JGit API 有时使用标准的 Java 异常（例如 `IOException` ），但是它也提供了大量 JGit 自己定义的异常类型（例如 `NoRemoteRepositoryException` 、 `CorruptObjectException` 和 `NoMergeBaseException` ）。

高层命令

底层 API 更加完善，但是有时将它们串起来以实现普通的目的非常困难，例如将一个文件添加到索引，或者创建一个新的提交。 为了解决这个问题， JGit 提供了一系列高层 API ，使用这些 API 的入口点就是 `Git` 类：

```
Repository repo;
// 构建仓库。。。
Git git = new Git(repo);
```

Git 类有一系列非常好的 构建器 风格的高层方法，它可以用来构造一些复杂的行为。 我们来看一个例子——做一件类似 `git ls-remote` 的事。

```
CredentialsProvider cp = new UsernamePasswordCredentialsProvider("username", "p4ssw0rd");
Collection<Ref> remoteRefs = git.lsRemote()
    .setCredentialsProvider(cp)
    .setRemote("origin")
    .setTags(true)
    .setHeads(false)
    .call();
for (Ref ref : remoteRefs) {
    System.out.println(ref.getName() + " -> " + ref.getObjectId().name());
```

这是一个 Git 类的公共样式，这个方法返回一个可以让你串连若干方法调用来设置参数的命令对象，当你调用 `.call()` 时它们就会被执行。 在这情况下，我们只是请求了 `origin` 远程的标签，而不是头部。 还要注意用于验证的 `CredentialsProvider` 对象的使用。

在 Git 类中还可以使用许多其它的命令，包括但不限于 `add`、`blame`、`commit`、`clean`、`push`、`rebase`、`revert` 和 `reset`。

拓展阅读

这只是 JGit 的全部能力的冰山一角。 如果你对这有兴趣并且想深入学习，在下面可以找到一些信息和灵感。

- JGit API 在线官方文档： http://download.eclipse.org/jgit/docs/latest/apidocs 。 这是基本的 Javadoc ，所以你也可以在你最喜欢的 JVM IDE 上将它们安装它们到本地。
- JGit Cookbook ： https://github.com/centic9/jgit-cookbook 拥有许多如何利用 JGit 实现特定任务的例子。
- http://stackoverflow.com/questions/6861881 指出了几个好的资源。

C. Git 命令

附录C. Git 命令

在这一整本书里我们介绍了大量的 Git 命令，并尽可能的通过讲故事的的方式来介绍它们，慢慢的介绍了越来越多的命令。但是这导致这些命令的示例用法都散落在在全书的各处。

在此附录中，我们会将本书中所提到过的命令都过一遍，并根据其用途大致的分类。我们会大致地讨论每个命的作用，指出其在本书中哪些章节使用过。

A3.1 设置与配置

设置与配置

有两个命令使用得最多了，从第一次调用 Git 到每天的日常微调及参考，这个两个命令就是：`config` 和 `help` 命令

git config

Git 做的很多工作都有一个默认方式。 对于绝大多数工作而言，你可以改变 Git 的默认方式，或者根据你的偏好来设置。 这些设置涵盖了所有的事，从告诉 Git 你的名字，到指定偏好的终端颜色，以及你使用的编辑器。 此命令会从几个特定的配置文件中读取和写入配置值，以便你可以从全局或者针对特定的仓库来进行设置。

本书的所有章节几乎都有用到 `git config` 命令。

在 初次运行 Git 前的配置 一节中，在开始使用 Git 之前，我们用它来指定我们的名字，邮箱地址和编辑器偏好。

在 Git 别名 一节中我们展示了如何创建可以展开为长选项序列的短命令，以便你不用每次都输入它们。

在 变基 一节中，执行 `git pull` 命令时，使用此命令来将 `--rebase` 作为默认选项。

在 凭证存储 一节中，我们使用它来为你的 HTTP 密码设置一个默认的存储区域。

在 关键字展开 一节中我们展示了如何设置在 Git 的内容添加和减少时使用的 smudge 过滤器 和 clean 过滤器。

最后，基本上 配置 Git 整个章节都是针对此命令的。

git help

`git help` 命令用来显示任何命令的 Git 自带文档。 但是我们仅会在此附录中提到大部分最常用的命令，对于每一个命令的完整的可选项及标志列表，你可以随时运行 `git help <command>` 命令来了解。

我们在 获取帮助 一节中介绍了 `git help` 命令，同时在 配置服务器 一节中给你展示了如何使用它来查找更多关于 `git shell` 的信息。

A3.2 获取与创建项目

获取与创建项目

有几种方式获取一个 Git 仓库。 一种是从网络上或者其他地方拷贝一个现有的仓库，另一种就是在一个目录中创建一个新的仓库。

git init

你只需要简单地运行 `git init` 就可以将一个目录转变成一个 Git 仓库，这样你就可以开始对它进行版本管理了。

我们一开始在 获取 Git 仓库 一节中介绍了如何创建一个新的仓库来开始工作。

在 远程分支 一节中我们简单的讨论了如何改变默认分支。

在 把裸仓库放到服务器上 一节中我们使用此命令来为一个服务器创建一个空的裸仓库。

最后，我们在 底层命令和高层命令 一节中介绍了此命令背后工作的原理的一些细节。

git clone

`git clone` 实际上是一个封装了其他几个命令的命令。 它创建了一个新目录，切换到新的目录，然后 `git init` 来初始化一个空的 Git 仓库， 然后为你指定的 URL 添加一个（默认名称为 `origin` 的）远程仓库（ `git remote add` ），再针对远程仓库执行 `git fetch` ，最后通过 `git checkout` 将远程仓库的最新提交检出到本地的工作目录。

`git clone` 命令在本书中多次用到，这里只列举几个有意思的地方。

在 克隆现有的仓库 一节中我们通过几个示例详细介绍了此命令。

在 在服务器上搭建 Git 一节中，我们使用了 `--bare` 选项来创建一个没有任何工作目录的 Git 仓库副本。

在 打包 一节中我们使用它来解包一个打包好的 Git 仓库。

最后，在 克隆含有子模块的项目 一节中我们学习了使用 `--recursive` 选项来让克隆一个带有子模块的仓库变得简单

虽然在本书的其他地方都有用到此命令，但是上面这些用法是特例，或者使用方式有点特别。

A3.3 快照基础

快照基础

对于基本的暂存内容及提交到你的历史记录中的工作流，只有少数基本的命令。

git add

`git add` 命令将内容从工作目录添加到暂存区（或称为索引（index）区），以备下次提交。 当 `git commit` 命令执行时，默认情况下它只会检查暂存区域，因此 `git add` 是用来确定下一次提交时快照的样子的。

这个命令对于 Git 来说特别的重要，所以在本书中被无数次的提及和使用。 我们将快速的过一遍一些可以看到的独特的用法。

我们在 跟踪新文件 一节中介绍并详细解释了 `git add` 命令。

然后，我们在 遇到冲突时的分支合并 一节中提到了如何使用它来解决合并冲突。

接下来，我们在 交互式暂存 一章中使用它来交互式的暂存一个已修改文件的特定部分。

最后，在 树对象 一节中我们在一个低层次中模拟了它的用法，以便你可以了解在这背后发生了什么。

git status

`git status` 命令将为你展示工作区及暂存区域中不同状态的文件。 这其中包含了已修改但未暂存，或已经暂存但没有提交的文件。 一般在它显示形式中，会给你展示一些关于如何在这些暂存区域之间移动文件的提示。

首先，我们在 检查当前文件状态 一节中介绍了 `status` 的基本及简单的形式。 虽然我们在全书中都有用到它，但是绝大部分的你能用 `git status` 做的事情都在这一章讲到了。

git diff

当需要查看任意两棵树的差异时你可以使用 `git diff` 命令。 此命令可以查看你工作环境与你的暂存区的差异（ `git diff` 默认的做法），你暂存区域与你最后提交之间的差异（ `git diff --staged` ），或者比较两个提交记录的差异（ `git diff master branchB` ）

首先，我们在 查看已暂存和未暂存的修改 一章中研究了 `git diff` 的基本用法，在此节中我们展示了如何查看哪些变化已经暂存了，哪些没有。

在 提交准则 一节中,我们在提交前使用 `--check` 选项来检查可能存在的空白字符问题。

在 确定引入了哪些东西 一节中,了解了使用 `git diff A...B` 语法来更有效地比较不同分支之间的差异。

在 高级合并 一节中我们使用 `-b` 选项来过滤掉空白字符的差异，及通过 `--theirs`、`--ours` 和 `--base` 选项来比较不同暂存区冲突文件的差异。

最后，在 开始使用子模块 一节中,我们使用此命令合 `--submodule` 选项来有效地比较子模块的变化。

git difftool

当你不想使用内置的 `git diff` 命令时。 `git difftool` 可以用来简单地启动一个外部工具来为你展示两棵树之间的差异。

我们只在 查看已暂存和未暂存的修改 一节中简单的提到了此命令。

git commit

`git commit` 命令将所有通过 `git add` 暂存的文件内容在数据库中创建一个持久的快照，然后将当前分支上的分支指针移到其之上。

首先，我们在 提交更新 一节中涉及了此命令的基本用法。 我们演示了如何在日常的工作流程中通过使用 `-a` 标志来跳过 `git add` 这一步，及如何使用 `-m` 标志通过命令行而不启动一个编辑器来传递提交信息。

在 撤消操作 一节中我们介绍了使用 `--amend` 选项来重做最后的提交。

在 分支简介，我们探讨了 `git commit` 的更多细节，及工作原理。

在 签署提交 一节中我们探讨了如何使用 `-S` 标志来为提交签名加密。

最后，在 提交对象 一节中，我们了解了 `git commit` 在背后做了什么，及它是如何实现的。

git reset

`git reset` 命令主要用来根据你传递给动作的参数来执行撤销操作。 它可以移动 `HEAD` 指针并且可选的改变 `index` 或者暂存区，如果你使用 `--hard` 参数的话你甚至可以改变工作区。 如果错误地为这个命令附加后面的参数，你可能会丢失你的工作，所以在使用前你要确定你已经完全理解了它。

首先，我们在 取消暂存的文件 一节中介绍了 `git reset` 简单高效的用法，用来对执行过 `git add` 命令的文件取消暂存。

在 重置揭密 一节中我们详细介绍了此命令，几乎整节都在解释此命令。

在 中断一次合并 一节中，我们使用 `git reset --hard` 来取消一个合并，同时我们也使用了 `git merge --abort` 命令，它是 `git reset` 的一个简单的封装。

git rm

`git rm` 是 Git 用来从工作区，或者暂存区移除文件的命令。 在为下一次提交暂存一个移除操作上，它与 `git add` 有一点类似。

我们在 移除文件 一节中提到了 `git rm` 的一些细节，包括递归地移除文件，和使用 `--cached` 选项来只移除暂存区域的文件但是保留工作区的文件。

在本书的 移除对象 一节中，介绍了 `git rm` 仅有的几种不同用法，如在执行 `git filter-branch` 中使用和解释了 `--ignore-unmatch` 选项。 这对脚本来说很有用。

git mv

`git mv` 命令是一个便利命令，用于移到一个文件并且在新文件上执行 `git add` 命令及在老文件上执行 `git rm` 命令。

我们只是在 移动文件 一节中简单地提到了此命令。

git clean

`git clean` 是一个用来从工作区中移除不想要的文件的命令。 可以是编译的临时文件或者合并冲突的文件。

在 清理工作目录 一节中我们介绍了你可能会使用 `clean` 命令的大量选项及场景。

A3.4 分支与合并

分支与合并

Git 有几个实现大部的分支及合并功能的实用命令。

git branch

`git branch` 命令实际上是某种程度上的分支管理工具。 它可以列出你所有的分支、创建新分支、删除分支及重命名分支。

Git 分支 一节主要是为 `branch` 命令来设计的，它贯穿了整个章节。 首先，我们在 分支创建 一节中介绍了它，然后我们在 分支管理 一节中介绍了它的其它大部分特性（列举及删除）。

在 跟踪分支 一节中，我们使用 `git branch -u` 选项来设置一个跟踪分支。

最后，我们在 Git 引用 一节中讲到了它在背后做一什么。

git checkout

`git checkout` 命令用来切换分支，或者检出内容到工作目录。

我们是在 分支切换 一节中第一次认识了命令及 `git branch` 命令。

在 跟踪分支 一节中我们了解了如何使用 `--track` 标志来开始跟踪分支。

在 检出冲突 一节中，我们用此命令和 `--conflict=diff3` 来重新介绍文件冲突。

在 重置揭密 一节中，我们进一步了解了其细节及与 `git reset` 的关系。

最后，我们在 HEAD 引用 一节中介绍了此命令的一些实现细节。

git merge

`git merge` 工具用来合并一个或者多个分支到你已经检出的分支中。 然后它将当前分支指针移动到合并结果上。

我们首先在 新建分支 一节中介绍了 `git merge` 命令。 虽然它在本书的各种地方都有用到，但是 `merge` 命令只有几个变种，一般只是 `git merge <branch>` 带上一个你想合并进来的一个分支名称。

我们在 派生的公开项目 的后面介绍了如何做一个 `squashed merge` （指 Git 合并时将其当作一个新的提交而不是记录你合并时的分支的历史记录。）

在 高级合并 一节中，我们介绍了合并的过程及命令，包含 `-Xignore-space-change` 命令及 `--abort` 选项来中止一个有问题的提交。

在 签署提交 一节中我们学习了如何在合并前验证签名，如果你项目正在使用 GPG 签名的话。

最后，我们在 子树合并 一节中学习了子树合并。

git mergetool

当你在 Git 的合并中遇到问题时，可以使用 `git mergetool` 来启动一个外部的合并帮助工具。

我们在 遇到冲突时的分支合并 中快速介绍了一下它，然后在 外部的合并与比较工具 一节中介绍了如何实现你自己的外部合并工具的细节。

git log

`git log` 命令用来展示一个项目的可达历史记录，从最近的提交快照起。 默认情况下，它只显示你当前所在分支的历史记录，但是可以显示不同的甚至多个头记录或分支以供遍历。 此命令通常也用来在提交记录级别显示两个或多个分支之间的差异。

在本书的每一章几乎都有用到此命令来描述一个项目的历史。

在 查看提交历史 一节中我们介绍了此命令，并深入做了研究。 研究了包括 `-p` 和 `--stat` 选项来了解每一个提交引入的变更，及使用 `--pretty` 和 `--online` 选项来查看简洁的历史记录。

在 分支创建 一节中我们使用它加 `--decorate` 选项来简单的可视化我们分支的指针所在，同时我们使用 `--graph` 选项来查看分叉的历史记录是怎么样的。

在 私有小型团队 和 提交区间 章节中，我们介绍了在使用 `git log` 命令时用 `branchA..branchB` 的语法来查看一个分支相对于另一个分支, 哪一些提交是唯一的。 在 提交区间 一节中我们作了更多介绍。

在 > 和 三点 章节中，我们介绍了 `branchA...branchB` 格式和 `--left-right` 语法来查看哪些仅其中一个分支。 在 合并日志 一节中我们还研究了如何使用 `--merge` 选项来帮助合并冲突调试，同样也使用 `--cc` 选项来查看在你历史记录中的合并提交的冲突。

在 引用日志 一节中我们使用此工具和 `-g` 选项 而不是遍历分支来查看 Git 的 `reflog` 。

在 搜索 一节中我们研究了 `-S` 及 `-L` 选项来进行来在代码的历史变更中进行相当优雅地搜索，如一个函数的历史。

在 签署提交 一节中，我们了解了如何使用 `--show-signature` 来为每一个提交的 `git log` 输出中，添加一个判断是否已经合法的签名的一个验证。

git stash

`git stash` 命令用来临时地保存一些还没有提交的工作，以便在分支上不需要提交未完成工作就可以清理工作目录。

储藏与清理 一整个章节基本就是在讲这个命令。

git tag

`git tag` 命令用来为代码历史记录中的某一个点指定一个永久的书签。 一般来说它用于发布相关事项。

我们在 打标签 一节中介绍了此命令及相关细节，并在 为发布打标签 一节实践了此命令。

我也在 签署工作 一节中介绍了如何使用 `-s` 标志创建一个 GPG 签名的标签，然后使用 `-v` 选项来验证。

A3.5 项目分享与更新

项目分享与更新

在 Git 中没有多少访问网络的命令，几乎所以的命令都是在操作本地的数据库。 当你想要分享你的工作，或者从其他地方拉取变更时，这有几个处理远程仓库的命令。

git fetch

`git fetch` 命令与一个远程的仓库交互，并且将远程仓库中有但是在当前仓库的没有的所有信息拉取下来然后存储在你本地数据库中。

我们开始在 从远程仓库中抓取与拉取 一节中介绍了此命令，然后我们在 远程分支 中看到了几个使用示例。

我们在 向一个项目贡献 一节中有几个示例中也都有使用此命令。

在 合并请求引用 我们用它来抓取一个在默认空间之外指定的引用，在 打包 中，我们了解了怎么从一个包中获取内容。

在 引用规格 章节中我们设置了高度自定义的 `refspec` 以便 `git fetch` 可以做一些跟默认不同的事情。

git pull

`git pull` 命令基本上就是 `git fetch` 和 `git merge` 命令的组合体，Git 从你指定的远程仓库中抓取内容，然后马上尝试将其合并进你所在的分支中。

我们在 从远程仓库中抓取与拉取 一节中快速介绍了此命令，然后在 查看远程仓库 一节中了解了如果你运行此命令的话，什么将会合并。

我们也在 用变基解决变基 一节中了解了如何使用此命令来来处理变基的难题。

在 检出冲突 一节中我们展示了使用此命令如何通过一个 URL 来一次性的拉取变更。

最后，我们在 签署提交 一节中我们快速的介绍了你可以使用 `--verify-signatures` 选项来验证你正在拉取下来的经过 GPG 签名的提交。

git push

`git push` 命令用来与另一个仓库通信，计算你本地数据库与远程仓库的差异，然后将差异推送到另一个仓库中。 它需要有另一个仓库的写权限，因此这通常是需要验证的。

我们开始在 推送到远程仓库 一节中介绍了 `git push` 命令。 在这一节中主要介绍了推送一个分支到远程仓库的基本用法。 在 推送 一节中，我们深入了解了如何推送指定分支，在 跟踪分支 一节中我们了解

了如何设置一个默认的推送的跟踪分支。 在 删除远程分支 一节中我们使用 `--delete` 标志和 `git push` 命令来在删除一个在服务器上的分支。

在 向一个项目贡献 一整节中，我们看到了几个使用 `git push` 在多个远程仓库分享分支中的工作的示例。

在 共享标签 一节中，我们知道了如何使用此命令加 `--tags` 选项来分享你打的标签。

在 发布子模块改动 一节中，我们使用 `--recurse-submodules` 选项来检查是否我们所有的子模块的工作都已经在推送子项目之前已经推送出去了，当使用子模块时这真的很有帮助。

在 其它客户端钩子 中我们简单的提到了 `pre-push` 挂钩（hook），它是一个可以用来设置成在一个推送完成之前运行的脚本，以检查推送是否被允许。

最后，在 引用规格推送 一节中，我们知道了使用完整的 refspec 来推送，而不是通常使用的简写形式。这对我们精确的指定要分享出去的工作很有帮助。

git remote

`git remote` 命令是一个是你远程仓库记录的管理工具。 它允许你将一个长的 URL 保存成一个简写的句柄，例如 `origin` ，这样你就可以不用每次都输入他们了。 你可以有多个这样的句柄，`git remote` 可以用来添加，修改，及删除它们。

此命令在 远程仓库的使用 一节中做了详细的介绍，包括列举、添加、移除、重命名功能。

几乎在此书的后续章节中都有使用此命令，但是一般是以 `git remote add <name> <url>` 这样的标准格式。

git archive

`git archive` 命令用来创建项目一个指定快照的归档文件。

我们在 准备一次发布 一节中，使用 `git archive` 命令来创建一个项目的归档文件用于分享。

git submodule

`git submodule` 命令用来管理一个仓库的其他外部仓库。 它可以被用在库或者其他类型的共享资源上。 `submodule` 命令有几个子命令, 如（ `add` 、 `update` 、 `sync` 等等）用来管理这些资源。

只在 子模块 章节中提到和详细介绍了此命令。

A3.6 检查与比较

检查与比较

git show

`git show` 命令可以以一种简单的人类可读的方式来显示一个 Git 对象。 你一般使用此命令来显示一个标签或一个提交的信息。

我们在 附注标签 一节中使用此命令来显示带注解标签的信息。

然后，我们在 选择修订版本 一节中，用了很多次来显示不同的版本选择将解析出来的提交。

我们使用 `git show` 做的最有意思的事情是在 手动文件再合并 一节中用来在合并冲突的多个暂存区域中提取指定文件的内容。

git shortlog

`git shortlog` 是一个用来归纳 `git log` 的输出的命令。 它可以接受很多与 `git log` 相同的选项，但是此命令并不会列出所有的提交，而是展示一个根据作者分组的提交记录的概括性信息

我们在 制作提交简报 一节中展示了如何使用此命令来创建一个漂亮的 changelog 文件。

git describe

`git describe` 命令用来接受任何可以解析成一个提交的东西，然后生成一个人类可读的字符串且不可变。 这是一种获得一个提交的描述的方式，它跟一个提交的 SHA-1 值一样是无歧义，但是更具可读性。

我们在 生成一个构建号 及 准备一次发布 章节中使用 `git describe` 命令来获得一个字符串来命名我们发布的文件。

A3.7 调试

调试

Git 有一些命令可以用来帮你调试你代码中的问题。 包括找出是什么时候，是谁引入的变更。

git bisect

`git bisect` 工具是一个非常有用的调试工具，它通过自动进行一个二分查找来找到哪一个特定的提交是导致 bug 或者问题的第一个提交。

仅在 二分查找 一节中完整的介绍了此命令。

git blame

`git blame` 命令标注任何文件的行，指出文件的每一行的最后的变更的提交及谁是那一个提交的作者。 当你要找那个人去询问关于这块特殊代码的信息时这会很有用。

只有 文件标注 一节有中提到此命令。

git grep

`git grep` 命令可以帮助在源代码中，甚至是你项目的老版本中的任意文件中查找任何字符串或者正则表达式。

只有 Git Grep 的章节中与提到此命令。

A3.8 补丁

补丁

Git 中的一些命令是以引入的变更即提交这样的概念为中心的，这样一系列的提交，就是一系列的补丁。这些命令以这样的方式来管理你的分支。

git cherry-pick

`git cherry-pick` 命令用来获得在单个提交中引入的变更，然后尝试将作为一个新的提交引入到你当前分支上。 从一个分支单独一个或者两个提交而不是合并整个分支的所有变更是非常有用的。

在 变基与拣选工作流 一节中描述和演示了 `Cherry picking`

git rebase

`git rebase` 命令基本是是一个自动化的 `cherry-pick` 命令。 它计算出一系列的提交，然后再以它们在其他地方以同样的顺序一个一个的 `cherry-picks` 出它们。

在 变基 一章中详细提到了此命令，包括与已经公开的分支的变基所涉及的协作问题。

在 替换 中我们在一个分离历史记录到两个单独的仓库的示例中实践了此命令，同时使用了 `--onto` 选项。

在 Rerere 一节中，我们研究了在变基时遇到的合并冲突的问题。

在 修改多个提交信息 一节中，我们也结合 `-i` 选项将其用于交互式的脚本模式。

git revert

`git revert` 命令本质上就是一个逆向的 `git cherry-pick` 操作。 它将你提交中的变更的以完全相反的方式的应用到一个新创建的提交中，本质上就是撤销或者倒转。

我们在 还原提交 一节中使用此命令来撤销一个合并提交。

A3.9 邮件

邮件

很多 Git 项目，包括 Git 本身，基本是通过邮件列表来维护的。 从方便地生成邮件补丁到从一个邮箱中应用这些补丁,Git 都有工具来让这些操作变得简单。

git apply

`git apply` 命令应用一个通过 `git diff` 或者甚至使用 GNU diff 命令创建的补丁。 它跟补丁命令做了差不多的工作，但还是有一些小小的差别。

我们在 应用来自邮件的补丁 一节中演示了它的使用及什么环境下你可能会用到它。

git am

`git am` 命令用来应用来自邮箱的补丁。特别是那些被 mbox 格式化过的。 这对于通过邮件接受补丁并将他们轻松地应用到你的项目中很有用。

我们在 使用 am 命令应用补丁 命令中提到了它的用法及工作流，包括使用 `--resolved`、 `-i` 及 `-3` 选项。

我们在 电子邮件工作流钩子 也提到了几条 hooks，你可以用来辅助与 `git am` 相关工作流。

在 邮件通知 一节中我们也将用此命令来应用 格式化的 GitHub的推送请求的变更。

git format-patch

`git format-patch` 命令用来以 mbox 的格式来生成一系列的补丁以便你可以发送到一个邮件列表中。

我们在 通过邮件的公开项目 一节中研究了一个使用 `git format-patch` 工具为一个项目做贡献的示例。

git imap-send

`git imap-send` 将一个由 `git format-patch` 生成的邮箱上传至 IMAP 草稿文件夹。 我们在通过邮件的公开项目 一节中见过一个通过使用 `git imap-send` 工具向一个项目发送补丁进行贡献的例子。

git send-email

`git send-mail` 命令用来通过邮件发送那些使用 `git format-patch` 生成的补丁。

我们在 通过邮件的公开项目 一节中研究了一个使用 `git send-email` 工具发送补丁来为一个项目做贡献的示例。

git request-pull

`git request-pull` 命令只是简单的用来生成一个可通过邮件发送给某个人的示例信息体。 如果你在公共服务器上有一个分支，并且想让别人知道如何集成这些变更，而不用通过邮件发送补丁，你就可以执行此命令的输出发送给这个你想拉取变更的人。

我们在 派生的公开项目 一节中演示了如何使用 `git request-pull` 来生成一个推送消息。

A3.10 外部系统

外部系统

Git 有一些可以与其他的版本控制系统集成的命令。

git svn

`git svn` 可以使 Git 作为一个客户端来与 Subversion 版本控制系统通信。 这意味着你可以使用 Git 来检出内容，或者提交到 Subversion 服务器。

Git 与 Subversion 一章深入讲解了此命令。

git fast-import

对于其他版本控制系统或者从其他任何的格式导入，你可以使用 `git fast-import` 快速地将其他格式映射到 Git 可以轻松记录的格式。

在 一个自定义的导入器 一节中深入讲解了此命令。

A3.11 管理

管理

如果你正在管理一个 Git 仓库，或者需要通过一个复杂的方法来修复某些东西，Git 提供了一些管理命令来帮助你。

git gc

`git gc` 命令在你的仓库中执行 “garbage collection” ，删除数据库中不需要的文件和将其他文件打包成一种更有效的格式。

此命令一般在背后为你工作，虽然你可以手动执行它-如果你想的话。 我们在维护 一节中研究此命令的几个示例。

git fsck

`git fsck` 命令用来检查内部数据库的问题或者不一致性。

我们只在 数据恢复 这一节中快速使用了一次此命令来搜索所有的漂流对象（dangling object）。

git reflog

`git reflog` 命令分析你所有分支的头指针的日志来查找出你在重写历史上可能丢失的提交。

我们主要在 引用日志 一节中提到了此命令，并在展示了一般用法，及如何使用 `git log -g` 来通过 `git log` 的输出来查看同样的信息。

我们同样在 数据恢复 一节中研究了一个恢复丢失的分支的实例。

git filter-branch

`git filter-branch` 命令用来根据某些规则来重写大量的提交记录，例如从任何地方删除文件，或者通过过滤一个仓库中的一个单独的子目录以提取出一个项目。

在 从每一个提交移除一个文件 一节中，我们解释了此命令，并探究了其他几个选项，例如 `--commit-filter` ， `--subdirectory-filter` 及 `--tree-filter` 。

在 Git-p4 和 TFS 的章节中我们使用它来修复已经导入到外部仓库。

A3.12 底层命令

底层命令

在本书中我们也遇到了不少底层的命令。

我们遇到的第一个底层命令是在 合并请求引用 中的 `ls-remote` 命令。我们用通过它来查看服务端的原始引用。

我们在 手动文件再合并、 Rerere 及 索引 章节中使用 `ls-files` 来查看暂存区的更原始的样子。

我们同样在 分支引用 一节中提到了 `rev-parse` 命令，它可以接受任意字符串，并将其转成一个对象的 SHA-1 值。

我们在 Git 内部原理 一章中对大部分的底层命令进行了介绍，这差不多正是这一章的重点所在。 我们尽量避免了在本书的其他部分使用这些命令。